JATKUVA INTEGROINTI

Java-ohjelmistotuotteelle

LAHDEN AMMATTIKORKEAKOULU
Tekniikan ala
Tietotekniikka
Ohjelmistotekniikka
Opinnäytetyö
Kevät 2011
Teemu Siltanen

Lahden ammattikorkeakoulu
Tietotekniikka

SILTANEN, TEEMU:

Jatkuva integrointi
Java-ohjelmistotuotteelle

Ohjelmistotekniikan opinnäytetyö, 38 sivua

Kevät 2011

TIIVISTELMÄ

Tämän opinnäytetyön aiheena on jatkuva integrointi Java-ohjelmistotuotteelle. Työn tarkoituksena on perehtyä ketterien menetelmien luomaan tarpeeseen saada nopeaa palautetta ohjelmiston kehityksestä sekä selvittää parhaat käytänteet jatkuvassa integroinnissa. Näiden tietojen avulla toteutetaan ohjelmointiympäristö, jossa on käytössä jatkuvaa integrointia suorittava palvelu. Tässä ympäristössä valmistetaan puhelinluettelo-ohjelma.

Työn toteuttamiseen käytetään yleisiä Java-ohjelmistokehityksessä käytössä olevia ohjelmistoja. Käännösympäristön luomiseen asennetaan CruiseControl, joka hyödyntää Ant-käännöstyökalua ohjelman kääntämiseen. Lisäksi muihin kehityksessä tarvittaviin osiin käytetään jUnit-yksikkötestauskehystä, Javadoc-dokumentointityökalua sekä Subversionia versionhallintajärjestelmänä.

Ohjelmistokehitys on muuttunut aikaisempaa nopeatahtisemmaksi, ja tämän vuoksi myös menetelmät ovat muuttuneet. Ketterien menetelmien tarkoituksena on keskittyä tuotteen valmistamiseen ja asiakkaan tarpeiden tyydyttämiseen. Tästä syystä yksi tällainen menetelmä Scrum perustuu lyhyisiin kehitysjaksoihin, joiden aikana tuotetaan julkaisukelpoisia parannuksia ohjelmistotuotteeseen. Jotta näissä lyhyissä jaksoissa voitaisiin saavuttaa riittävän laadukas parannus, on testattava riittävästi. Tähän tarpeeseen vastaa testivetoinen kehitys, jossa luodaan testit ennen varsinaisen ohjelman kirjoittamista.

Jatkuvan integroinnin parhaiden käytänteiden soveltaminen käytännössä helpottaa ohjelman kehittämistä, koska jokaisen muutoksen jälkeen suoritettava automaattinen käännös antaa mahdollisuuden havaita aikaisessa vaiheessa mahdolliset ristiriidat eri osien integroinnissa. Työn toteutus käyttäen annettuja työkaluja onnistui hyvin, ja saatu kokemus jatkuvan integroinnin käytöstä tukee sen käyttöä ohjelmistokehityksessä.

Avainsanat: Scrum, testivetoinen kehitys, jatkuva integrointi

Lahti University of Applied Sciences
Degree Programme in Software Engineering

SILTANEN, TEEMU:

Continuous Integration
Case: Java Software

Bachelor's Thesis in Software Engineering, 38 pages

Spring 2011

ABSTRACT

The topic of this thesis is continuous integration for a Java software product. The purpose was to study the needs created by agile methods to get rapid feedback from software development and to examine best practices of continuous integration. A software development environment which includes a service implementing continuous integration was created according to this information. In this environment a phonebook software product was designed.

To accomplish the task, common software used in Java software development was used. CruiseControl was used for establishing the build environment. It utilizes the Ant build tool for building the software. Other tools needed in the development were the jUnit testing framework, Javadoc documentation tool and Subversion as the version control system.

Software development has become faster paced than earlier and that is why the methods have also changed. The purpose of agile methods is to focus on creating product and satisfying the needs of the client. Because of this, one such method, Scrum, is based on short development periods in which publishable improvements to a software product are created. To achieve good enough improvement in these short periods, there has to be enough testing. An answer to this need is test-driven development where tests are written before coding the actual program.

Implementing the best practices of continuous integration in practice makes software development easier as automated build run after every change gives an oppourtunity to notice possible conflicts in integrating parts of program together at an early phase. The implementation part of the thesis succeeded well and the results support using continuous integration in software development.

Key words: Scrum, test-driven development, continuous integration

SISÄLLYS

1 JOHDANTO 1

2 KETTERIÄ MENETELMIÄ 2

2.1 Scrum 3

2.1.1 Prosessi 3

2.1.2 Roolit 5

2.2 Testivetoinen kehitys 6

2.2.1 Hyödyt 8

2.2.2 Perussykli 9

2.2.3 Heikkoudet 11

2.2.4 jUnit-yksikkötestauskehys 11

3 JATKUVA INTEGROINTI 13

3.1 Parhaat käytänteet 13

3.1.1 Versionhallinnan ylläpito 14

3.1.2 Kääntämisen automatisointi 14

3.1.3 Testien automatisointi 15

3.1.4 Muutokset päivittäisiä 15

3.1.5 Kaikki versiot integroidaan 16

3.1.6 Lyhyt käännöksen kesto 16

3.1.7 Oikeanlainen testausympäristö 17

3.1.8 Informaation jakaminen 17

3.2 CruiseControl 18

3.2.1 Ant-käännöstyökalu 18

3.2.2 Asennus 20

3.2.3 Projektin konfiguroiminen 20

3.2.4 Tulosten tarkastelu 23

3.2.5 Javadoc-dokumentointityökalu 25

4 OHJELMISTOPROJEKTIN TYÖVAIHEET 28

4.1 Kehitysympäristö 28

4.2 Integrointipalvelimen määrittäminen 29

4.3 Projektin kuvaus 32

4.4 Toteutus 33

4.5 Dokumentointi 35

5 YHTEENVETO 37

LÄHTEET 39

1 JOHDANTO

Ohjelmistoalalla siirrytään jatkuvasti nopeampaan kehitystahtiin markkinavoimien pyrkiessä saamaan valmistettava tuote mahdollisimman nopeasti myytäväksi. Perinteisistä raskaista ja heikosti muutoksiin sopeutuvista kehitysmenetelmistä ollaan luopumassa ja tilalle ovat tulleet lukuisat ketterät menetelmät, kuten Scrum, testivetoinen kehitys ja näihin liittyvä jatkuva integrointi. Näiden avulla voidaan ottaa huomioon perinteistä nopeammin kesken ohjelmistokehitysprosessin ilmenevät muutostarpeet sekä helpottaa markkinointia tarjoamalla ohjelmasta toimiva kehitysversio jo varhaisessa vaiheessa.

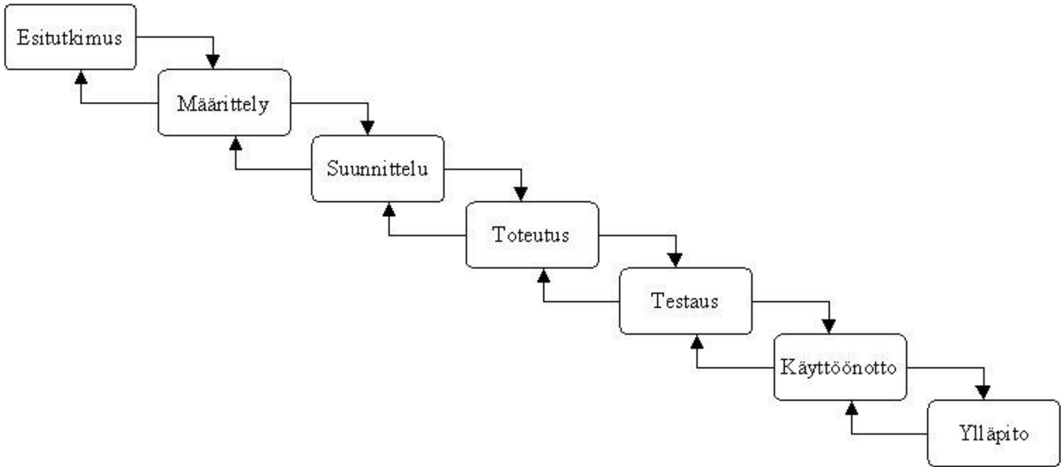
Työ tehdään Lahden ammattikorkeakoulun tekniikan alan tietotekniikan koulutusohjelmalle. Ohjelmistotekniikan opettajia työskentelee tekniikan alalla viisi, joiden lisäksi on muiden koulutusalojen ja suuntautumisvaihtoehtojen opettajia.

Työn tavoitteena on perehtyä jatkuvan integroinnin vaatimuksiin sekä löytää parhaat käytännöt, joita jatkuvassa integroinnissa tulisi soveltaa, jotta ohjelmistokehitysprosessiin syntyisi mahdollisimman vähän yleisrasitetta. Lisäksi pyritään selvittämään ketterien menetelmien, kuten testivetoisen kehityksen, käyttöä yhdessä jatkuvan integroinnin kanssa. Tarkoituksena on toteuttaa ympäristö, jossa kehittäjien versionhallintaan tekemät muutokset aiheuttavat integrointipalvelimen kääntämään ohjelmiston automaattisesti ja suorittamaan yksikkötestit. Lisäksi tämän palvelimen tulisi välittää kehittäjille tieto käännöksen ja testien tuloksista ja tarjota syntyneet jakelutiedostot ja dokumentaatiot saataville. Ympäristön toimivuuden demonstroimiseksi tehdään puhelinluettelo-ohjelma, joka toimii hajautetusti käyttäen tekstitiedostoa numeroiden lähteenä.

Työssä keskitytään ohjelmistokehitykseen Java-ympäristössä käyttäen sille suunniteltuja työkaluja. Jatkuvaa integrointia varten käytetään CruiseControl-integrointikehystä, joka on tehty suorittamaan automaattisia käännöksiä. Itse käännöstä varten käytetään Java-kääntäjää sekä Ant-käännöstyökalua yhdessä CruiseControl:n kanssa. Tarvittavien yksikkötestien luomiseen ja suorittamiseen käytetään jUnit-yksikkötestauskehystä, joka on suunniteltu Java-ohjelmistoja varten.

2 KETTERIÄ MENETELMIÄ

Ohjelmistotekniikka on vielä suhteellisen nuori ala, joka kehittyy eteenpäin kiivaaseen tahtiin jatkuvasti. Samalla ohjelmistokehitysmenetelmät muuttuvat nopeammiksi ja tarpeisiin mukautuviksi. Perinteiset kehitysmenetelmät, kuten vesiputousmalli, jäävät taakse tehokkaampien menetelmien tarpeessa. Vesiputousmallia ei voida täysin kirjaimellisesti soveltaa käytännössä ohjelmistokehityksessä, koska osa vaatimuksista, jotka vesiputousmallin mukaisesti tulee kirjata ylös projektin aluksi, ilmenee vasta projektin aikana. Kuviossa 1 on esitetty vesiputousmallin kulku. (Haikala & Märijärvi 2006, 41.)



KUVIO 1. Vesiputousmalli

Vesiputousmallia korvaamaan kehitettiin inkrementaalisia ja iteratiivisia malleja, joissa ohjelmistoa kehitetään muutosten sarjana. Ohjelmistoa kehitetään luomalla siihen toimiva ominaisuus toisensa jälkeen, jolloin kesken projektin ilmenevät tarpeet on helpompi sisällyttää ohjelmistoon. Viime aikoina niin kutsutut ketterät menetelmät, jotka perustuvat lyhyisiin iteraatioihin, ovat tulleet yleisiksi. Näissä menetelmissä projektin kehitys ja testaus on jatkuvaa ja uusien ominaisuuksien lisääminen on mahdollista myös kesken projektin. Iteraatiot pitävät kukin sisällään ohjelmistoprojektin työvaiheet suunnittelusta dokumentointiin. Jokaisen iteraation lopuksi on tarkoitus saada valmiiksi julkaisukelpoinen tuote. (Haikala & Märijärvi 2006, 47.)

Vuonna 2001 merkittävät ketterien kehitysmenetelmiä eteenpäin kehittävät henkilöt kokoontuivat yhteen, ja tämän tuloksena syntyi niin sanottu Agile Manifesto, jota pidetään nykyisin ketterien menetelmien perusmääritelmänä. Tuo manifesti pitää sisällään neljä perusperiaatetta, jotka kuvaavat ketterien menetelmien ideaa. Nuo teesit ovat:

> *Me etsimme parempia keinoja ohjelmistojen kehittämiseen tekemällä sitä itse ja auttamalla siinä muita.*
>
> *Tässä työssämme olemme päätyneet arvostamaan*
>
> ***Yksilöitä ja vuorovaikutusta** enemmän kuin prosesseja ja työkaluja*
>
> ***Toimivaa sovellusta** enemmän kuin kokonaisvaltaista dokumentaatiota*
>
> ***Asiakasyhteistyötä** enemmän kuin sopimusneuvotteluita*
>
> ***Muutokseen reagoimista** enemmän kuin suunnitelman noudattamista*

(Agile Manifesto 2001.)

Näiden manifestissa ilmaistujen periaatteiden pohjalta on kehitetty uusia ketteriä menetelmiä, jotka pyrkivät täyttämään nuo tavoitteet. Yksi näistä menetelmistä on Scrum.

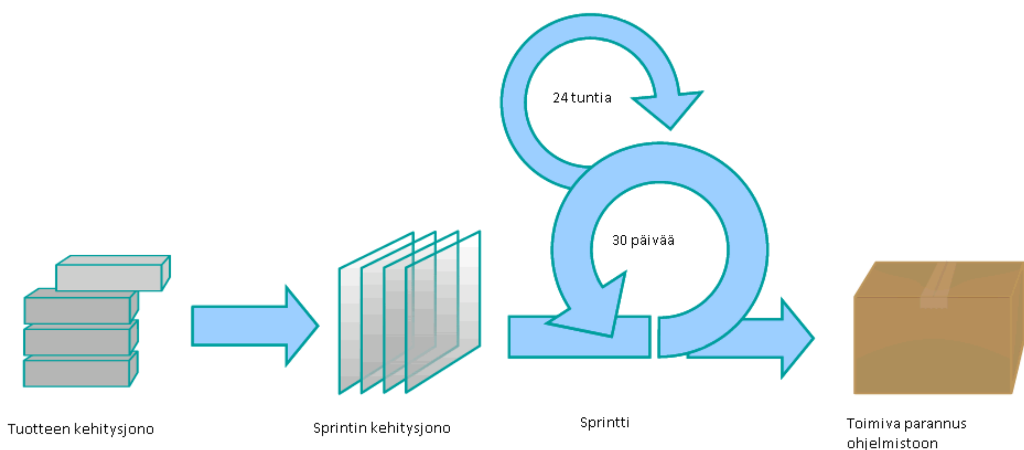
2.1 Scrum

Scrum yleiskuvaltaan on iteratiivinen ja inkrementaalinen projektinhallintamalli, jota käytetään laajalti ohjelmistoalalla. Scrumin monista säännöistä johtuen monet kehittäjät ovat päätyneet ottamaan käyttöönsä vain heille parhaiten soveltuvat osat Scrumista.

2.1.1 Prosessi

Scrum määrittelee käytettäväksi rooleja, jotka ovat scrummaster, tuoteomistaja sekä kehitystiimi. Scrum-prosessi alkaa tuotteen kehitysjonon luomisesta. Tämän

tekeminen voidaan aloittaa pitämällä julkaisun suunnittelukokous, jossa määritellään tavoitteet ja suunnitelma, jonka koko organisaatio ymmärtää. Jos tällainen kokous pidetään, voidaan sen perusteella valvoa tuotteen kehityskaaren aikana sen edistymistä ja tavoitteiden täyttymistä. Jos tätä kokousta ei kuitenkaan pidetä, puuttuvasta suunnittelutiedosta syntyy este, joka scrumtiimin tulee poistaa lisämäällä tarvittavat asiat tuotteen kehitysjonoon. (Schwaber & Sutherland 2010.)



KUVIO 2. Scrum-prosessi (Wikipedia 2011)

Scrum pitää sisällään pyrähdyksiä, eli niin sanottuja sprinttejä, joihin sisältyy sprintin suunnittelu, varsinainen kehitystyö sekä sprintin katselmointi. Sprintin sisällä tehdään vain niitä asioita, jotka on määritetty tehtäväksi kyseisen sprintin aikana. Sen sisältöä ei voida muokata kesken, ellei tuoteomistajalla ole jokin tärkeä syy tehdä toisin. Sprintin kesto on maksimissaan 30 päivää, ja ne seuraavat toisiaan ilman taukoja niiden välissä, kuten kuviossa 2 oleva yleiskuva Scrum-prosessista havainnollistaa. (Schwaber & Sutherland 2010.)

Jokaisen sprintin aluksi pidetään suunnittelukokous, jossa suunnitellaan alkava kehitysjakso. Tämä kokous koostuu kahdesta osasta, jossa toisessa määritellään se, mitä tullaan tekemään sprintin aikana ja toisessa osassa suunnitellaan, miten nuo asiat aiotaan suorittaa. Scrumtiimi saa valita itse, kuinka paljon työtä he ottavat tehtäväksi tuotteen kehitysjonosta sprintin aikana tehtäväksi. Kun tuo päätös on tehty, se pysyy tuon kyseisen sprintin tavoitteena. Kokouksen jälkimmäisessä osassa valitut tuotteen kehitysjonon kohteet pyritään jakamaan osiin, jotka voi-

daan toteuttaa alle päivässä, ja näistä muodostetaan sprintin kehitysjono. (Schwaber & Sutherland 2010.)

Sprintin sisällä jokaisena päivänä kehitystiimi pitää lyhyen kokouksen, jossa tarkastellaan kunkin jäsenen edellisen päivän tuotosta, kyseisen päivän tavoitetta sekä mahdollisia esteitä tämän tavoitteen saavuttamiseksi. Kokousten tarkoituksena on parantaa kehitystiimin ymmärrystä kehityksen etenemisestä suhteessa sprintin tavoitteeseen. (Schwaber & Sutherland 2010.)

Jokaisen sprintin lopuksi pidetään katselmointikokous, jossa tarkastellaan sitä, miten sprintin tavoitteet on saavutettu. Kehitystiimi esittelee tekemänsä työn ja selostaa mitä mahdollisia ongelmia sen tekemisen aikana ilmeni ja miten ne ratkaistiin. Jos sprintin aikana saatu valmiiksi kaikkia sille suunniteltuja kohteita, nämä siirtyvät takaisin tuotteen kehitysjonoon, josta ne otetaan jonkin toisen sprintin työlistalle. Tämän jälkeen tuoteomistaja selventää tuotteen kehitysjonon sen hetkistä tilannetta ja tuotteen valmistumisajankohta-arvion. Kokouksen lopuksi tarkastellaan ilmi tulleiden tietojen vaikutusta tuleviin sprintteihin ja hankitaan näin pohjatietoa seuraavan sprintin suunnittelukokoukseen. (Schwaber & Sutherland 2010.)

2.1.2 Roolit

Scrum määrittelee käytettäväksi tiimejä, joissa on jokaiselle oma roolinsa. Näitä rooleja on kolme: Scrummaster, tuoteomistaja sekä kehitystiimi. (Schwaber & Sutherland 2010.)

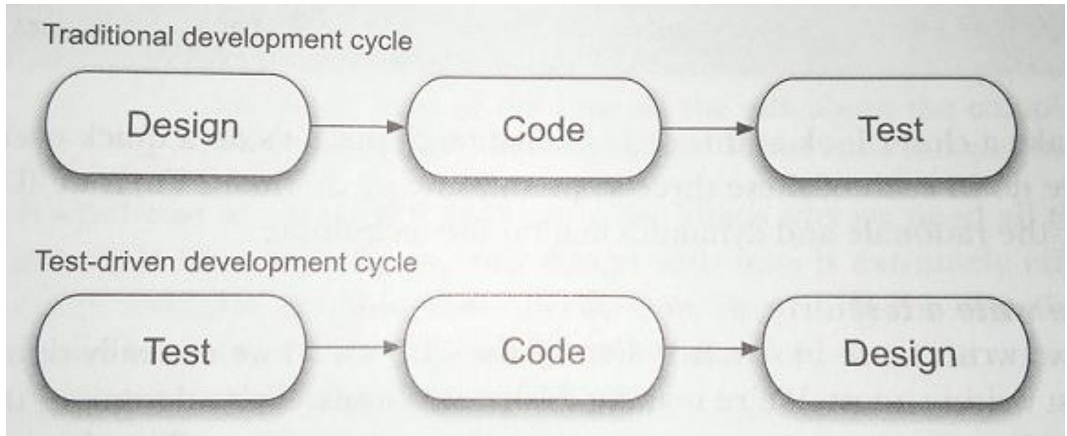
Scrummasterin tehtävänä on varmistaa, että kaikki scrumtiimin jäsenet noudattavat Scrumin käytäntöjä ja sääntöjä. Hänen tehtävänään on myös opettaa muita tiimin jäseniä ymmärtämään Scrumin ominaisuuksia ja tapoja toimia. Yksi hänen tehtävistään on auttaa muita tekemään parhaansa käytössä olevassa ympäristössä ja poistaa kehitystä mahdollisesti estävät tekijät. Scrummaster on siis ryhmää johtava sekä palveleva jäsen. (Schwaber & Sutherland 2010.)

Tuoteomistaja on henkilö, joka huolehtii valmistettavan tuotteen kehitysjonosta sekä valvoo tehtävän työn laatua. Hän ylläpitää kehitysjonoa ja pitää huolen siitä, että kaikki ovat tietoisia siitä, millä töillä on suurin prioriteetti, jolloin kaikki tietävät, mitä tullaan tekemään. Tuoteomistajan päätökset ovat näkyviä kaikille ja jotta hän voisi onnistua tehtävässään, tiimin jäsenien tulisi mukautua hänen päätöksiinsä. (Schwaber & Sutherland 2010.)

Kehitystiimin jäsenet tekevät tuotteen kehitysjonossa olevista kohteista jokaisen sprintin päätteeksi julkaisukelpoisen tuoteparannuksen. Kehitystiimin tulee olla monitaitoinen, jotta he voivat saada aikaiseksi toimivan parannuksen tuotteeseen. Tiimin koko on viidestä yhdeksään henkeä, ja se koostuu eri asioihin erikoistuneista osaajista, jotka kaikki kuitenkin tekevät kaikkia tarvittavia töitä. Kehitystiimi on itseorganisoituva ja päättää itse, miten tuotteen kehitysjonossa olevat kohteet valmistetaan toimivaksi tuoteparannukseksi. (Schwaber & Sutherland 2010.)

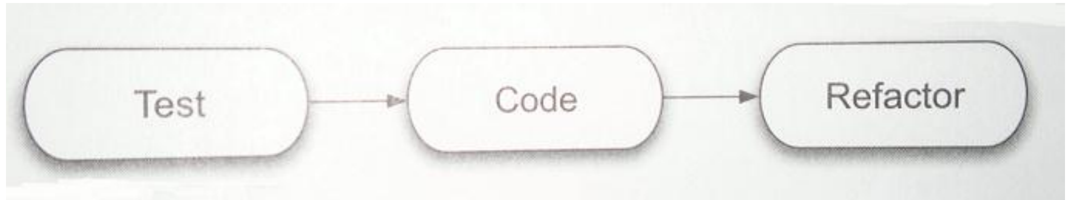
2.2 Testivetoinen kehitys

Ohjelmistotuotteen markkinoijilla on tarve saada mahdollisimman nopeasti käsiinsä toimiva versio ohjelmasta, jota he voivat esitellä. Tämän mahdollistamiseksi on tärkeää pitää koko ohjelman kehityskaaren läpi saatavilla testattu ja toimiva versio, jonka sitä tarvitsevat voivat saada käyttöönsä. Myös ketterien menetelmien, kuten Scrumin, nopeassa tahdissa lyhyiden syklien aikana kehitettävä koodi tarvitsee jatkuvaa testausta. Perinteinen lähestymistapa koodin suunnitteluun ja kirjoittamiseen ennen testausta, mutta testivetoisessa tämä käännetään päinvastaiseen järjestykseen. Kuten kuviosta 3 käy ilmi tässä kehitysmenetelmässä tehdään ensin testi, jonka jälkeen sille kirjoitetaan toteutus. (Koskela 2008, 15.)



KUVIO 3. Testivetoisen kehityksen sykli verrattuna perinteiseen (Koskela 2008, 15)

Testivetoinen kehitys perustuu siis yksinkertaiseen ja lyhyeen kiertoon, jossa ohjelmiston kehittäjä kirjoittaa ensin testin metodille, jonka hän kirjoittaa jälkikäteen. Luonnollisesti tämä kirjoitettu testi ei mene läpi ennen kuin sen täyttävä ohjelmanosa on kirjoitettu, mutta juuri tämä onkin testivetoisen kehityksen perimmäinen idea. Testin epäonnistuttua kehittäjä kirjoittaa metodin, joka suorittaa testin läpi menemiseksi vaadittavat asiat yksinkertaisimmalla mahdollisella tavalla. Tämän jälkeen testiä muutetaan lisäämällä testitapauksia, jonka jälkeen juuri tehty toteutus ei enää täytä vaatimuksia. Tämän jälkeen kehittäjä parantaa metodia siten, että testi menee jälleen läpi eli refaktoroi koodia. Tätä kiertoa jatketaan kunnes ollaan tyytyväisiä metodin toimintaan, jonka jälkeen voidaan siirtyä seuraavan metodin toteuttamiseen käyttäen samaa menetelmää. Yksittäisen metodien kirjoittamisessa käytettävän menetelmän voi havainnollistaa kuviossa 4 näkyvällä kaaviolla. (Koskela 2008, 16.)



KUVIO 4. Testaus, koodaus ja refaktorointi -menetelmä (Koskela 2008, 16)

Käytännön testivetoisessa kehityksessä on järkevää ottaa käyttöön jokin yksikkötestauskehys, joka helpottaa testien kirjoittamista ja näin vähentää siihen kuluvaa niin sanottua hukka-aikaa eli aikaa, jota ei käytetä varsinaisen ohjelman koodaa-

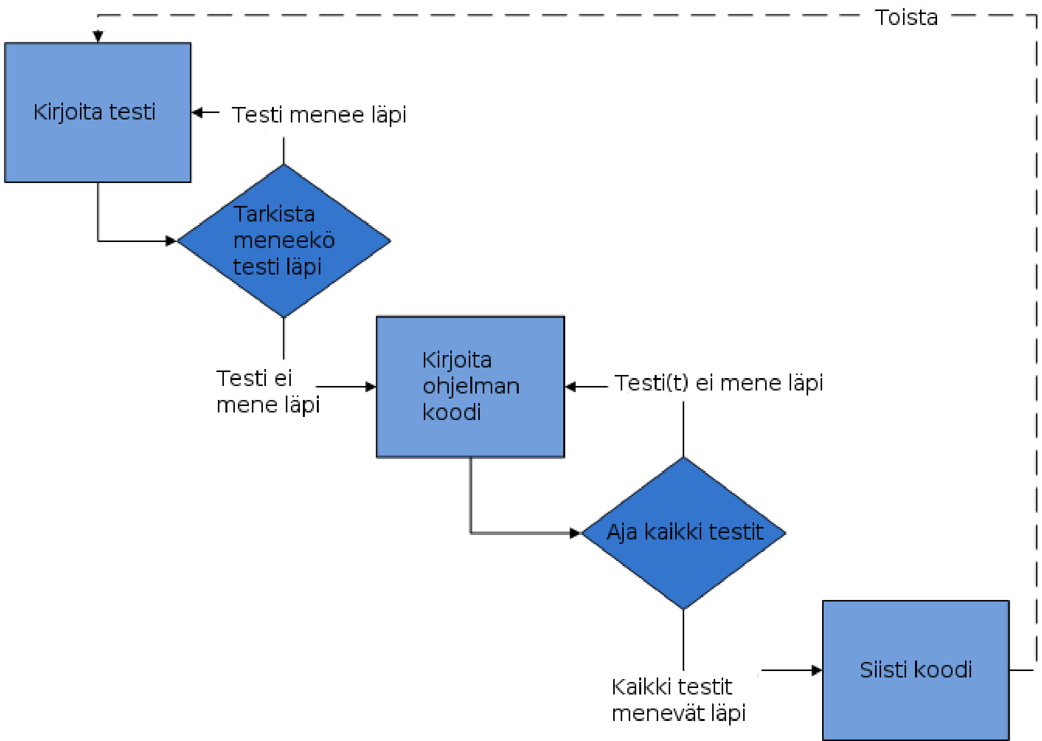
miseen. Näitä on tarjolla useita erilaisia ja monille eri kielille. Yksi yleisesti Java-ohjelmoinnissa käytössä oleva kehys on jUnit, jota käytetään tässä työssä. (Wikipedia 2011.)

2.2.1 Hyödyt

Hyötynä perinteiseen ohjelmistokehitykseen verrattuna testivetoisesta kehityksestä testit muodostavat eräänlaisen suojaverkon, joka auttaa pitämään projektin edetessä syntyvien ongelmien paikantamiseen ja korjaamiseen kuluvan ajan mahdollisimman pienenä. Koska jokaiselle metodille on kirjoitettu sitä testaava testi, voidaan tehdä ohjelmistoon melko suuriakin muutoksia myös projektin myöhemmässä vaiheessa. Mahdollisesti esiin nousevat ristiriidat muun ohjelmiston koodin kanssa voidaan paikallistaa hyvin nopeasti testien ansiosta ja tehdä tarvittavat muutokset ohjelman toimintaan saattamiseksi. (Llopis 2005.)

Toinen testivetoisesta kehityksestä koituva hyöty on ohjelman modulaarisuus. Koska jokainen funktio tehdään erikseen, ja jokaista testataan erillisenä osana, päädytään todennäköisesti lopputulokseen, jossa ohjelmiston osia voidaan käyttää uudelleen useassa eri paikassa ja uusien ominaisuuksien lisääminen jälkikäteen on helpompaa. Myös koodin kääntäminen tapahtuu nopeammin kuin isojen kokonaisuuksien, joita perinteisessä ohjelmoinnissa syntyy helpommin. (Llopis 2005.)

Testivetoisen kehityksestä koituva hyöty on myös kehittäjän virheiden löytämiseen kuluneen ajan pieneneminen. Kun kehittäjä tekee muutoksia koodiin ja haluaa nähdä sen toimivuuden, hän kääntää sen ja ajaa testit. Näistä hän näkee nopeasti, toimiiko ohjelma niin kuin sen pitäisi ilman ohjelman käsin tehtävää ajoa kokeillen mahdollisia syötteitä. Jos testit eivät mene läpi, ohjelmassa on jotain vialla, jolloin kehittäjä tekee koodiin tarpeelliset muutokset ja ajaa testit uudelleen. Sykli on tässä nopeaa, ja virheet on mahdollista korjata suhteellisen nopeasti. (Llopis 2005.)



KUVIO 5. Testivetoisen kehityksen sykli (Wikipedia 2011)

2.2.2 Perussykli

Testivetoisen kehityksen sykli on yksinkertainen, kuten voidaan nähdä kuviosta 5. Kaikki alkaa testin kirjoittamisesta. Tyypillisesti testi on lyhyt ja yksinkertainen, mutta ohjelmoijan taitojen kasvaessa ja ohjelman monimutkaistuessa saatetaan joutua tekemään testejä myös isommille kokonaisuuksille, vaikka tavoitteena on pitää osat mahdollisimman pieninä. Esimerkkinä voidaan kirjoittaa testi luokalle, jonka metodi laskee kaksi lukua yhteen. Kuviossa 6 on näkyvissä tällainen testi useamman refaktorointi kerran jälkeen. Testin ensimmäisessä versiossa testSum()-metodi on sisältänyt ainoastaan yhden vertailun, rivillä 4 näkyvän tarkistuksen.

```
public class TestAdder {
    public void testSum() {
        Adder adder = new AdderImpl();
        assert(adder.add(1, 1) == 2);
        assert(adder.add(1, 2) == 3);
        assert(adder.add(2, 2) == 4);
        assert(adder.add(0, 0) == 0);
        assert(adder.add(-1, -2) == -3);
        assert(adder.add(-1, 1) == 0);
        assert(adder.add(1234, 988) == 2222);
    }
}
```

KUVIO 6. Yksikkötestiesimerkki

Seuraavana vaiheena testin kirjoittamisen jälkeen ajetaan testi ja varmistetaan, että se ei mene läpi, jolloin testi olisi kirjoitettu virheellisesti. Tämän jälkeen kirjoitetaan yksinkertaisin ratkaisu, joka läpäisee juuri kirjoitetun testin. Yksinkertaisin ratkaisu testin ensimmäiselle versiolle on antaa paluuarvoksi vakio 2, mutta useampien refaktorointi kertojen jälkeen päädytään jo yleiskäytöisempään ja monimutkaisempaan ratkaisuun. Kuviossa 6 olevasta testistä voidaan päätellä, että tarvitaan rajapinta Adder ja sen toteuttava luokka AdderImpl, joiden yksinkertainen toteutus näkyy kuvioissa 7 ja 8.

```
public interface Adder {
    public int add(int a, int b);
}
```

KUVIO 7. Rajapinta Adder

```
public class AdderImpl implements Adder {
    public int add(int a, int b) {
        return a + b;
    }
}
```

KUVIO 8. Rajapinnan toteuttava luokka AdderImpl

Kun on saatu kirjoitettua koodi, jonka pitäisi toteuttaa testin vaatimat toiminnot, ajetaan kaikki testit ja tarkistetaan jääkö jokin testeistä, joko juuri kirjoitettu tai jokin aikaisemmin luoduista, menemättä läpi. Jos jokin testeistä ei mene läpi, tehdään tarvittavat muutokset ja ajetaan testit uudelleen. Tätä tehdään, kunnes kaikki

testit menevät läpi. Joissain tapauksissa saattaa olla nopeampaa poistaa tehty implementaatio ja kirjoittaa se kokonaan uudestaan sen sijaan, että yrittää muokata olemassa olevasta toimivan. Kun testit on läpäisty, siistitään koodista ylimääräiset ja siirrytään tekemään seuraavalle funktiolle testiä ja aloitetaan kierto alusta.

2.2.3 Heikkoudet

Yksikkötestien läpäisemä tuotos ei kuitenkaan välttämättä ole virheetön, sillä testit varmistavat ainoastaan sen, että ne kyseiset asiat ja funktiot toimivat niin kuin niiden on tarkoitus toimia. Niitä tulisikin käyttää yhteistoiminnassa muiden ohjelmistokehityksessä käytettävien testaustapojen, kuten integrointitestauksen, kanssa. Joissakin asioissa testien tekeminen ja ajaminen saattaa olla aikaa vievää ja työlästä, kuten esimerkiksi käyttöliittymissä ja verkkoliikennettä vaativissa operaatioissa. (Wikipedia 2011.)

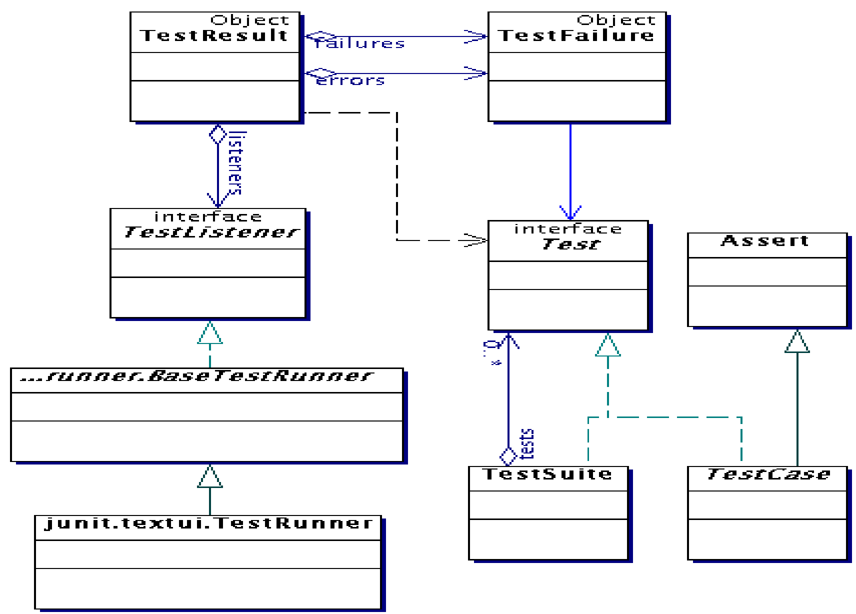
Heikkouksiin testivetoisessa kehityksessä kuuluu se, että testien kirjoittamiseen ja ylläpitämiseen kuluu aikaa. Jos kaikki kehittäjät ovat kuitenkin sitoutuneita tähän kehitystapaan ja osaavat kirjoittaa tehokkaita testejä kohtuullisessa ajassa, tämä kulunut aika saadaan pois projektin loppuvaiheessa normaalisti tapahtuvasta testauksesta, koska iso osa testeistä on jo valmiina. Lisäksi jos testit ajetaan automaattisesti, säästyy kehittäjältä tämäkin aika. (Wikipedia 2011.)

2.2.4 jUnit-yksikkötestauskehys

Kuten aikaisemmin mainittiin, testien valmistamisen nopeuttamiseksi voidaan käyttää yksikkötestauskehystä. Seuraavaksi tarkastellaan hieman tarkemmin jUnit-kehyksen toimintaa tässä tarkoituksessa.

Kuviossa 9 on kuvattu jUnit-kehyksen käyttämä rakenne. jUnit käyttää testien ajamiseen testisarjoja (TestSuite), jotka kootaan samaan paikkaan, josta ne voidaan ajaa yhtä aikaa. Tämä testisarja palauttaa tiedot testien onnistumisesta testikohtaisesti, jotka jUnit kerää yhdeksi raportiksi. Yksi testisarja voi sisältää useita

testitapauksia (TestCase), jotka käyttävät jUnit-kehyksen tarjoamia vertailumetodeja (Assert), joita on luotu mahdollisimman moneen eri käyttötarkoitukseen.



KUVIO 9. jUnit-kehyksen käyttämä rakenne (Hong 2005)

Testien ajamisen helpottamiseksi testisarjoille voidaan määritellä ennen testien ajoa suoritettavat toiminnot metodilla setUp(). Vastaavasti testien ajon jälkeen tehtävät toiminnot voidaan määritellä metodilla tearDown(). Näiden käyttäminen nopeuttaa testien kirjoittamista, koska esimerkiksi jonkin luokan olion alustus voidaan määritellä yhdessä paikassa kaikille testisarjan testitapauksille. (Hong 2005.)

jUnit-yksikkötestauskehys on liitetty useaan editoriin, joiden avulla voidaan generoida kirjoitetuista metodeista testisarja, joka sisältää kyseisille metodeille testitapaukset, joihin tarvitsee kirjoittaa haluttu sisältö. Tämä nopeuttaa testien kirjoittamista, ja lisäksi testit voidaan useassa tapauksessa myös ajaa suoraan editorin kautta, jolloin tähänkään ei kulu ylimääräistä aikaa.

3 JATKUVA INTEGROINTI

Jatkuva integrointi on ohjelmistokehitysmenetelmä, jossa kehitystiimin jokainen jäsen integroi luomansa ohjelman ominaisuudet kokonaisuuteen yleensä vähintään kerran päivässä. Integraation toimivuus tarkastetaan automatisoidun käännöksen avulla, johon sisältyy tehtyjen testien ajaminen. Tämän tavoitteena on vähentää virheiden löytämiseen ja korjaamiseen kuluvaa aikaa, koska virheet pyritään korjaamaan mahdollisimman nopeasti. Tällöin tehdyt muutokset ovat vielä tuoreessa muistissa, ja virheiden etsimisalue on pienempi kuin koko projektin lopuksi suoritettavassa integraatiossa löytyvissä virheissä. (Fowler 2006.)

Periaatteellinen työnkulku käytettäessä jatkuvaa integrointia ohjelmistoprojektissa alkaa normaalisti siitä, että kehittäjä hakee uusimman toimivan version versionhallintajärjestelmästä omalle työkoneelleen. Tämän jälkeen toteutetaan työn alla oleva ominaisuus testeineen ja tarkistetaan sen toimivuus omalla kehityskoneella kääntämällä se sekä ajamalla luodut testit. Sen tultua valmiiksi lähetetään se takaisin versionhallintaan, josta jatkuvaa integrointia suorittava työkalu hakee sen ja suorittaa sille määritellyt toiminnot. Tyypillisesti näihin kuuluvat ainakin kääntö sekä testien ajaminen. Suorittamalla nämä myös kehityskoneen ulkopuolella voidaan havaita mahdolliset virheet, joita syntyy käännettäessä ja testattaessa koko ohjelmistoa yhdessä, joita kehittäjä ei välttämättä ole huomannut tai osannut ottaa huomioon aikaisemmin. Jos tämä käännös ei mene läpi, kehittäjälle ilmoitetaan siitä välittömästi, jolloin hän voi tehdä tarvittavat korjaukset mahdollisimman nopeasti. (Fowler 2006.)

3.1 Parhaat käytänteet

Jotta jatkuvasta integroinnista saadaan suurin hyöty, on suositeltavaa noudattaa joitakin hyviksi todettuja käytäntöjä. Seuraavaksi tarkastellaan joitakin tällaisia käytänteitä tarkemmin eriteltynä.

3.1.1 Versionhallinnan ylläpito

Ohjelmistoprojekti koostuu monista eri tiedostoista, joita tarvitaan toimivan ohjelman luomiseksi. Näiden ylläpitämiseksi etenkin usean kehittäjän projektissa on järkevää ottaa käyttöön jokin versionhallintaohjelma, joka pitää tiedostot samassa paikassa kaikkien kehittäjien saatavilla. Versionhallintaan tulisi lisätä kaikki toimivan ohjelman kääntämiseen tarvittavat tiedostot, jolloin jokainen kehittäjä voi testata toimivuutta suoraan ottamalla nämä tiedostot versionhallinnasta itselleen. Myös mahdolliset ulkopuoliset ohjelman tarvitsemat tiedostot ja kirjastot tulisi sisällyttää versionhallintaan, jotta kehitystyö olisi mahdollisimman jouhevaa. (Fowler 2006.)

On huomionarvoista, että vaikka yleisimmät versionhallintajärjestelmät tukevat ohjelmiston jakamista useampaan eri haaraan, on suositeltavaa käyttää näitä rajoitetusti. Liian monien eri haarojen käyttö saattaa johtaa ongelmiin myöhemmin kun näitä liitetään päähaaraan. Järkeviä haaroja voisivat olla julkaistavasta tuotteesta löydettyjen virheiden korjausta varten luotu haara sekä mahdollisesti jonkin asian testaamiseksi luotu väliaikainen haara. (Fowler 2006.)

3.1.2 Kääntämisen automatisointi

Toimivan ohjelman tuottamiseen sisältyy usein monenlaisia tehtäviä kääntämisen lisäksi, kuten esimerkiksi tiedostojen siirteleminen paikasta toiseen. Useimmat näistä toiminnoista voidaan automatisoida ja säästää näin aikaa. Näiden toimintojen suorittamiseksi on kehitetty useita työkaluja, kuten Unix-ympäristössä usein käytettävä make sekä Java-ympäristöä varten kehitetty Ant, joilla voidaan tehdä suurin osa kääntämisen ohessa tarvittavista tehtävistä. (Fowler 2006.)

Suuren ohjelmiston kääntäminen kokonaisuudessaan saattaa viedä pitkän ajan, joten hyvän käännöstyökalun tulisi mahdollistaa kääntäminen myös pienemmissä osissa. Useat käännöstyökalut osaavat tarkistaa mitkä tiedostot ovat muuttuneet ja kääntää ohjelma uudestaan vain niiltä osin kuin on tarpeellista. Käännöksiä voi-

daan myös tehdä eri tarpeiden mukaisesti, esimerkiksi yksi käännös testikoodien kanssa ja toinen ilman niitä. (Fowler 2006.)

3.1.3 Testien automatisointi

Ohjelman kääntyminen suoritettavaksi ohjelmaksi vaatii sen menemistä kääntäjän läpi. Kääntäjä osaa ilmoittaa mahdollisista syntaksivirheistä koodissa, mutta ohjelma ei ole virheetön vaikka kääntäjä ei ilmoittaisikaan virheistä. Tämän vuoksi tarvitaan testejä, jotka tarkistavat ohjelman loogista toimintaa eli sitä, että se tekee asiat, joita sen on tarkoitus tehdä. (Fowler 2006.)

Kaikkien testien suorittaminen tulisi olla yksinkertaista, ei monia komentoja vaativaa. Testitkään eivät löydä kaikkia ohjelmiston virheitä, mutta ne auttavat kuitenkin löytämään monia sellaisia virheitä, joista pelkkä kääntäjä ei osaa ilmoittaa käännöksen yhteydessä. (Fowler 2006.)

3.1.4 Muutokset päivittäisiä

Jotta jatkuva integrointi olisi mahdollista, kaikkien kehittäjien tulisi päivittää muokkauksensa versionhallintaan mahdollisimman usein, vähintään kerran päivässä. Näin tekemällä mahdolliset ristiriidat ohjelman toisien osien kanssa tulevat ilmi nopeasti, ja ne voidaan korjata saman tien, jolloin ei muodostu isoa sumaa yhtäaikaista integrointia vaativia muutoksia eri ohjelman osiin. (Fowler 2006.)

Mitä useammin versionhallintaan tehdään muutoksia, sitä pienempi on alue, jolta mahdollisia virheitä tarvitsee etsiä ja korjata. Näin säästyy aikaa, kun asiat ovat vielä kehittäjän tuoreessa muistissa ja on helpompaa vertailla tehtyjä muutoksia edelliseen versioon verrattuna. Joissain tapauksissa saattaa olla nopeampaa vain palata takaisin edelliseen versioon ja tehdä lisätty ominaisuus uudestaan. (Fowler 2006.)

3.1.5 Kaikki versiot integroidaan

Vaikka kehittäjät kääntävät ja testaavat ohjelmaa jokaisen muutoksen yhteydessä omalla kehityskoneellaan, käännöksen toimivuus tulisi tarkistaa erillisellä integrointikoneella, jonka ympäristö on asetettu projektille soveltuvaksi. Näin tekemällä minimoidaan kehittäjien koneiden eroista johtuvat ongelmat. Versionhallintaan tehdyn muutoksen toimivuus tulisi tarkistaa tällä erillisellä koneella ajetuista käännöksestä ja testipatterista. (Fowler 2006.)

Integrointikoneella suoritettaviin toimiin on kaksi vaihtoehtoa: manuaalinen suoritus tai automaattisesti palvelimen suorittamat tehtävät. Manuaalisessa vaihtoehdossa kehittäjä menee muutoksen tehtyään integrointikoneelle suorittamaan käännöksen ja testit seuraten, miten ne onnistuvat. Integrointipalvelimen tapauksessa palvelin tarkkailee versionhallintaan tehtäviä muutoksia ja suorittaa automaattisesti muutoksen tapahtuessa käännöksen sekä testit ja mahdolliset muut tarvittavat tehtävät. Suoritettuaan nämä palvelin ilmoittaa miten näiden kävi tavalla, joka vaihtelee käytettävästä palvelimen ohjelmistosta. Tätä tarkoitusta varten on tehty lukuisia eri ohjelmia, muun muassa laajalti käytössä oleva CruiseControl. (Fowler 2006.)

3.1.6 Lyhyt käännöksen kesto

Jatkuvan integroinnin perimmäinen tarkoitus on antaa kehittäjille nopeasti palautetta tehdyistä muutoksista. Jos yksittäinen käännös kuitenkin vie paljon aikaa, ei tätä tavoitetta voida saavuttaa, koska se suoritetaan jokaisen versionhallintaan tehtävän päivityksen jälkeen. Tämän vuoksi kääntämiseen käytettävä aika tulisi saattaa minimiinsä. (Fowler 2006.)

Usein suurin hidaste löytyy käännöksen yhteydessä tehtävistä testeistä, erityisesti jos nuo testit sisältävät hakuja ulkopuolisista lähteistä, kuten esimerkiksi tietokannoista. Tällaisissa tapauksissa voidaan jakaa käännös useampaan osaan. Jokaisen päivityksen jälkeen ajetaan vain toiminnan varmistamisen kannalta välttämättömät testit, jotka eivät kestä kohtuuttoman kauan. Loput testit voidaan ajaa erikseen

harvempaan tahtiin kuin nämä pääasialliseen käännöksen liittyvät testit, jolloin pystytään pitämään kiinni jatkuvan integroinnin periaattesta saada tieto kriittisistä virheistä mahdollisimman nopeasti. (Fowler 2006.)

3.1.7 Oikeanlainen testausympäristö

Luotua ohjelmaa tulisi testata siinä ympäristössä, jossa sitä tullaan käyttämään sen valmistuttua, jotta saataisiin kiinni mahdollisimman monet siinä olevat virheet ennen toimitusta asiakkaalle. Pienetkin eroavaisuudet testausympäristössä saattavat jättää huomiotta virheitä, jotka häiritsevät ohjelman toimintaa loppukäyttäjällä. (Fowler 2006.)

Monesti on kallista tai jopa mahdotonta testata ohjelmaa kaikissa loppukäyttöympäristöissä, etenkin tehtäessä työpöytäsovelluksia, koska mahdollisia kombinaatioita on lähes rajattomasti. Tästä huolimatta tulisi pyrkiä jäljittelemään lopullista käyttöympäristöä niin hyvin kuin on mahdollista, jotta vähennetään riskiä julkaista toimimaton ohjelma. (Fowler 2006.)

3.1.8 Informaation jakaminen

Tuotettavan ohjelman viimeisimmän version tulisi olla kaikkien kehitykseen osallisten saatavilla läpi sen elinkaaren. Tämä auttaa esittelemään tuotetta ja löytämään siitä mahdollisesti virheellisesti luodut ominaisuudet tai siitä puuttuvat oleelliset piirteet, jotka eivät ole tulleet aikaisemmin esille. (Fowler 2006.)

Tämän lisäksi jokaisen kehittäjän tulisi olla selvillä jokaisen käännöksen lopputuloksesta ja tehdyistä muutoksista. Jos käytössä on jokin ohjelmisto, joka suorittaa jatkuvan integroinnin, se todennäköisesti tarjoaa kanavan tämän tekemiselle jollakin kätevällä tavalla, kuten esimerkiksi web-sivuston kautta. Vaikka käytössä ei olisikaan mitään tuollaista ohjelmaa, kehittäjien tulisi sopia tapa, jolla tieto välitetään kaikille. Näin kaikki ovat selvillä siitä, miten projekti edistyy ja mitä ongelmia on kohdattu. (Fowler 2006.)

3.2 CruiseControl

On olemassa lukuisia eri ohjelmistoja auttamaan jatkuvan integroinnin käytännön toteutusta, jotka tarjoavat samankaltaisia perustoimintoja. Tässä työssä tarkastellaan tarkemmin avoimen lähdekoodin CruiseControl:ia.

CruiseControl on Java-pohjainen kehys jatkuvaa integrointia varten, jonka mukana tulee Ant-kääntötyökalu, mutta joka tukee myös muita käännöstyökaluja. Siitä on kehitetty myös .NET-ympäristöä varten oma versionsa, mutta se ei eroa merkittävästi muilta osin Javaa varten tehdystä versiosta.

3.2.1 Ant-käännöstyökalu

Koska CruiseControl itsessään ei ole kääntäjä eikä näin ollen kykene suorittamaan tarvittavia toimintoja yksinään, tarvitaan jokin käännöstyökalu sen lisäksi. CruiseControl sisältää Ant-käännöstyökalun, jota voidaan käyttää Java-projektien kääntämiseen, joten seuraavaksi tarkastellaan hieman sen käyttöä ja toimintaa.

Ant:n käyttö perustuu XML-formaatissa kirjoitettuun tiedostoon, jonka sisään määritellään tehtävät, jotka Ant suorittaa. Kun Ant suoritetaan, kutsutaan tätä tiedostoa joka ilman parametreja tai annetaan parametrina kyseisestä tiedostosta löytyvä kohde, joka halutaan suorittaa. Usein nämä tiedostot nimetään build.xml-nimellä, joka nopeuttaa esimerkiksi avoimen lähdekoodin ohjelmiston kääntämistä, koska ei tarvitse etsiä käännöstiedostoa tietämättä nimeä. (Apache Ant 2011.)

Jokainen tällainen käännöstiedosto sisältää vähintään yhden projektin (project) ja yhden suoritettavan kohteen (target). Kohteiden sisälle kirjoitetaan toiminnot, jotka halutaan suoritettavan. Kuviossa 10 on esimerkki yksinkertaisesta käännöstiedostosta, jolla suoritetaan lähes jokaisessa Java-projektissa tarvittavat komennot. (Apache Ant 2011.)

```
1  <project name="MyProject" default="dist" basedir=".">
2      <property name="src" location="src"/>
3      <property name="build" location="build"/>
4      <property name="dist"  location="dist"/>
5
6      <target name="init">
7          <mkdir dir="${build}"/>
8      </target>
9      <target name="compile" depends="init">
10         <javac srcdir="${src}" destdir="${build}"/>
11     </target>
12     <target name="dist" depends="compile">
13         <mkdir dir="${dist}/lib"/>
14         <jar
15             jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
16             basedir="${build}" />
17     </target>
18     <target name="clean">
19         <delete dir="${build}"/>
20         <delete dir="${dist}"/>
21     </target>
22 </project>
```

KUVIO 10. Ant:n käännöstiedosto

Kuvion 10 rivillä 1 oleva <project>-elementille voidaan määritellä kolme attribuuttia: nimi, oletuksena suoritettava kohde sekä hakemistopolku, josta kyseisen projektin tehtävät suoritetaan. Käännöstiedoston selkeyttämiseksi ja tietojen muokkauksen helpottamiseksi on hyvä tapa kirjoittaa usein toistuvat nimet, kuten hakemistopolut, property-muuttujiin. Kuviossa 10 riveillä 2 - 4 on sijoitettu tällaisiin muuttujiin lähdekoodikansio sekä kansiot käännöstuotoksille. (Apache Ant 2011.)

Suoritettavat kohteet nimetään <target>-elementin attribuutilla name (kuvion 10 rivi 6). Tämän annetun nimen perusteella näitä kohteita voidaan kutsua sekä käännöstiedoston sisällä että Ant-käännöstä suoritettaessa. Kohteet voivat olla riippuvaisia toisista, jolloin jonkin kohteen suorittaminen edellyttää toisen kohteen suorittamista ensin. Kohde voidaan asettaa riippuvaiseksi toisesta attribuutilla depends, jonka arvoksi annetaan halutun kohteen nimi. Kuvion 10 esimerkissä on määritetty jar-tiedoston luova kohde dist riippuvaiseksi kääntämisen suorittavasta kohteesta compile, joka puolestaan on riippuvainen alustuskohteesta init (kuvion 10 rivit 6, 9 ja 12). (Apache Ant 2011.)

Kohteiden sisään kirjoitetaan tehtävät, jotka ovat käännöstiedoston varsinainen suoritettava osa. Nämä tehtävät kirjoitetaan aina samaan muotoon, jossa ensin on tehtävän nimi ja sen perässä jokin määrä attribuutteja. Tehtävän nimi voi olla jokin Ant-syntaksiin sisäänrakennetuista tai itsemääritelty tehtävä. Kuvion 10 esimerkissä on käytetty sisäänrakennettuja tehtäviä, kuten mkdir, javac, jar sekä delete (rivit 7, 10, 13 ja 19). (Apache Ant 2011.)

3.2.2 Asennus

CruiseControl on kehitetty sekä Windows- että Linux-ympäristöön, ja sen toiminta on ympäristöstä riippumatta suurin piirtein samanlaista. CruiseControl:in asentaminen on myös yhtä helppoa kummassakin ympäristössä. Tästä eteenpäin tarkastellaan CruiseControl:in käyttöönottoa Windows-ympäristössä.

Asentaminen alkaa binäärijakelun hakemisesta CruiseControl:in kotisivuilta. Tämän jälkeen puretaan haettu jakelu palvelimelle haluttuun paikkaan. Lopuksi vaaditaan vain CruiseControl:in käynnistäminen puretusta kansiosta löytävästä cruisecontrol.bat nimisestä tiedostosta. Näin CruiseControl on asennettu sen oletusasetuksilla. (CruiseControl 2011.)

3.2.3 Projektin konfiguroiminen

Kaikki CruiseControl:lle määriteltävät tehtävät kirjoitetaan config.xml-tiedostoon, joka sijaitsee oletuksena samassa kansiossa kuin CruiseControl:n käynnistävä cruisecontrol.bat. Tämä tiedosto noudattaa standardia XML-kielen syntaksia käyttäen ennalta määrättyjä elementtien ja attribuuttien nimiä. Näitä elementtejä on määritelty 126 erilaista, mutta liitännäisten käyttö sallii myös muiden elementtien käytön.

Usein lähellekään kaikkia näistä elementeistä ei tarvitse käyttää projektin konfiguroimiseen vaan riittää muutamien lähes jokaisesta projektista löytyvien elementtien käyttö. Kuviossa 11 on esimerkki yksinkertaisesta konfigurointitiedostosta,

joka sisältää oleellisimmat elementit. Juurielementti <cruisecontrol> (kuvion 11 rivi 1) tulee löytyä konfigurointitiedostosta. Tämän elementin alle sijoitetaan muut elementit, yksi tai useampi <project>-elementti sekä mahdollisia yleisiä asetustietoja sisältäviä elementtejä (CruiseControl 2011.)

```
1  <cruisecontrol>
2      <project name="esimerkki">
3          <listeners>
4              <currentbuildstatuslistener
5                  file="logs/${project.name}/status.txt"/>
6          </listeners>
7
8          <bootstrappers>
9              <antbootstrapper
10                 anthome="apache-ant-1.7.0"
11                 buildfile="projects/${project.name}/build.xml"
12                 target="clean" />
13         </bootstrappers>
14
15         <modificationset quietperiod="30">
16             <filesystem folder="projects/${project.name}"/>
17         </modificationset>
18
19         <schedule interval="300">
20             <ant
21                 anthome="apache-ant-1.7.0"
22                 buildfile="projects/${project.name}/build.xml"/>
23         </schedule>
24
25         <log>
26             <merge dir="projects/${project.name}/target/test-results"/>
27         </log>
28
29         <publishers>
30             <onsuccess>
31                 <artifactspublisher
32                     dest="artifacts/${project.name}"
33                     file="projects/${project.name}/target/${project.name}.jar"/>
34             </onsuccess>
35         </publishers>
36     </project>
37 </cruisecontrol>
```

KUVIO 11. Esimerkki config.xml

<project>-elementti (kuvion 11 rivi 2) sisältää tiedon siitä mitä tulee tehdä, milloin se tulee suorittaa ja miten tehdyt tehtävät raportoidaan. <project>-elementille voidaan antaa kaksi attribuuttia, name ja buildafterfailed. Name-attribuuttiin sijoitetaan projektin nimi, ja tämä on pakko asettaa. Buildafterfailed-attribuutti ei ole pakollinen, mutta sillä voidaan määritellä yrittääkö CruiseControl kääntää projektin uudelleen epäonnistuneen käännöksen jälkeen vaikka versionhallintaan ei ole tehty muutoksia. Oletusarvoisesti attribuutin arvo on true, eli CruiseControl yrittää kääntää uudelleen vaikka muutoksia ei ole tehty. (Koskela 2004.)

<bootstrappers>-elementti (kuvion 11 rivi 8) sisältää tiedon tehtävistä, jotka suoritetaan joka kerta ennen kuin varsinainen käännös suoritettaisiin. Tätä voidaan käyttää muun muassa jonkin tiedoston päivittämiseen versionhallinnasta ennen käännöstä tai jonkin projektille tärkeän tehtävän suorittamiseen, jotta käännös toimisi toivotulla tavalla. Näitä ennen käännöstä suoritettavia toimintoja voidaan määritellä vapaavalintainen määrä, ja ne ovat toisistaan täysin riippumattomia. (CruiseControl 2011.)

<project>-elementin sisällä ainoastaan kaksi elementtiä on pakollisia ja ensimmäinen niistä on <modificationset>-elementti (kuvion 11 rivi 15). Tämän elementin perusteella CruiseControl päättelee, onko käännös tarpeellista suorittaa. Tälle elementille voidaan asettaa kaksi attribuuttia, joista kumpikaan ei ole pakollinen. Ensimmäinen attribuutti on requiremodification, joka kertoo sen, tarvitseeko projekti tehtyjä muokkauksia ennen kuin käännös suoritetaan. Oletuksena sen arvo on true, eli muokkaus vaaditaan ennen käännöstä. Toinen mahdollinen attribuutti on quietperiod, joka kertoo sen ajan, jonka CruiseControl vaatii ennen käännöksen suorittamista, jonka aikana ei ole tehty muutoksia käännettäviin tiedostoihin. Oletuksena tämän attribuutin arvo on 60, joka on usein riittävän pitkä aika varmistamaan, ettei vain osa muutoksista ole saapunut perille. Tämän elementin sisään määritellään paikka, josta muutoksia etsitään. Kuvion 11 esimerkissä rivillä 16 muutoksia etsitään paikallisesta tiedostojärjestelmästä, mutta usein seurataan myös jotain versionhallintajärjestelmää. (Koskela 2004.)

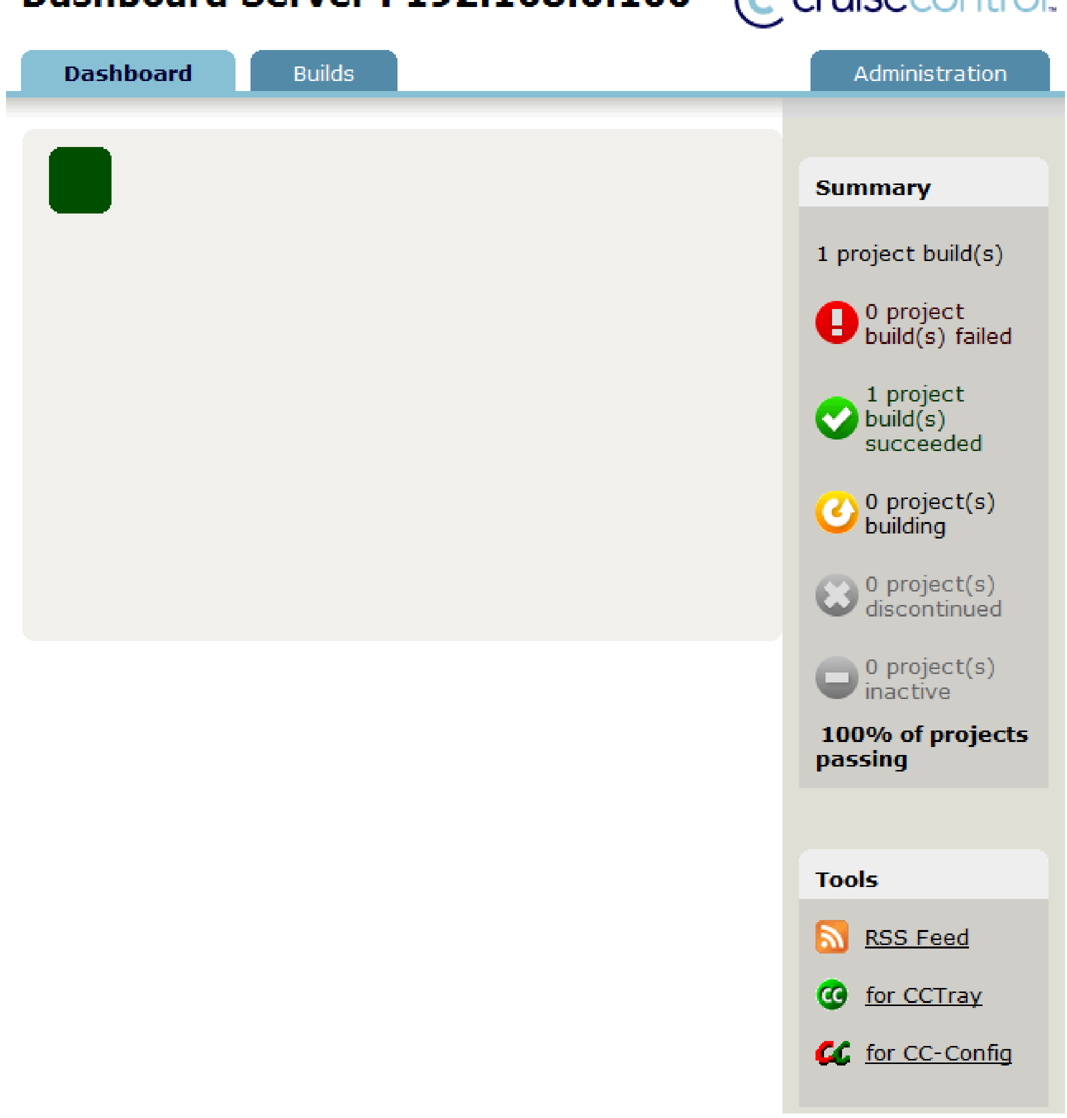
Toinen pakollinen elementti on <schedule> (kuvion 11 rivi 19). Tämän elementin sisään määritellään tehtävät, jotka suoritetaan varsinaisen käännöksen yhteydessä. Elementille on mahdollista määritellä yksi attribuutti, interval. Tämä määrittelee, kuinka usein käännös yritetään tehdä. Oletusarvona on 300 eli käännös yritetään suorittaa viiden minuutin välein. <schedule>-elementin lapsielementit määrittelevät kääntäjän ja sen suoritettavat tehtävät. Näitä lapsielementtejä on 11 erilaista, joista vain yhtä voidaan käyttää käännöstä kohden. Kuvion 11 esimerkissä on käytetty käännnöstyökaluna Ant:ia (rivi 20). Yksi <schedule>-elementin lapsielementeistä on <pause>, jota voidaan käyttää pysäyttämään käännös esimerkiksi viikonlopuksi tai muuksi ajaksi, jolloin muutoksia ei tehdä. (CruiseControl 2011.)

Seuraava yleinen elementti on <log> (kuvion 11 rivi 25). Tämän tarkoituksena on kerätä kertynyt tieto yhteen paikkaan, josta sitä voidaan hyödyntää. Tälle voidaan määritellä attribuutti dir, joka kertoo, mistä CruiseControl etsii käännöshistoriaa kyseiselle projektille. Oletuksena tämän arvo on ”logs/[projectname]”, jonne CruiseControl tallentaa projektin käännöshistorian, jos toisin ei ole määritelty. Lapsielementti <merge> määrittelee, mitä muuta näihin lokeihin tulisi lisätä, esimerkiksi testaustulokset kuten kuvion 11 esimerkissä rivillä 26. (CruiseControl 2011.)

<publishers>-elementti (kuvio 11 rivi 29) sisältää toiminnot, jotka suoritetaan sen jälkeen, kun käännöstä on yritetty suorittaa. Kuvion 11 esimerkissä siirretään käännöksen onnistuessa syntynyt jar-tiedosto toiseen kansioon, mutta tämän elementin alla on mahdollista suorittaa monia muitakin toimintoja. Esimerkiksi yksi hyödyllinen käyttötapa on lähettää kehittäjälle sähköpostiviesti käännöksen lopputuloksesta tai vain käännöksen epäonnistuessa. (CruiseControl 2011.)

3.2.4 Tulosten tarkastelu

CruiseControl tarjoaa kaksi eri tapaa tarkastella projektien tilaa ja syntyneitä lokeja. Ensimmäinen näistä on web-sivusto, jonka CruiseControl luo automaattisesti käytössään olevien tietojen perusteella. Tältä sivustolta voidaan nähdä yleiskatsaus kaikkiin projekteihin sekä näiden yksityiskohtaisemmat käännöstiedot. Oletuksena tämä sivusto löytyy tätä palvelua suorittavan palvelimen osoitteesta http://localhost:8080/dashboard/, jossa localhost on tuon palvelimen IP-osoite. Kuviossa 12 on tuon sivuston etusivu, jossa näkyy yhteenveto kaikista CruiseControl:n piiriin liitetyistä projekteista. Tässä esimerkissä ei ole kuin yksi projekti, jota kuvaa vihreä neliö sen merkiksi, että käännös on mennyt virheettömästi läpi. Vastaavasti, jos käännös ei olisi mennyt läpi, tuo neliö olisi punainen. (CruiseControl 2011.)

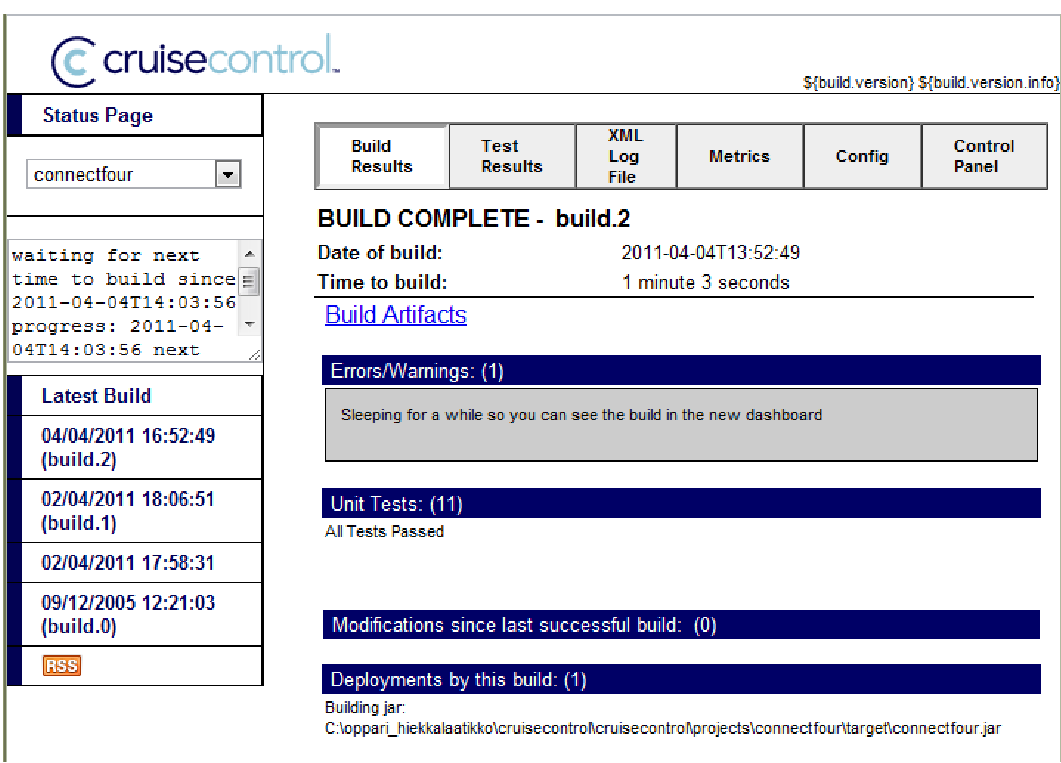


KUVIO 12. CruiseControl:n raporttisivusto

Tältä sivustolta on mahdollista myös ladata kaikki käännöksen seurauksena syntyneet tuotokset sekä tarkastella testien tuloksia ja muita lokimerkintöjä. Sivuston kautta on myös mahdollista muokata joitakin konfigurointitietoja kuten esimerkiksi sitä, kuinka usein käännöstä yritetään suorittaa. Tarvittaessa käännös voidaan myös pakottaa suoritettavaksi välittömästi tämän sivuston kautta. (CruiseControl 2011.)

Toinen tapa tarkastella syntyneitä tietoja on JavaServer Pages (JSP) -sivun kautta, jonka CruiseControl luo automaattisesti tietojensa pohjalta osoitteeseen http://localhost:8080/cruisecontrol, jossa localhost on palvelimen IP-osoite. Tä-

män sivun kautta voidaan tehdä samat asiat kuin edellä mainitun sivustonkin kautta, mutta lisäksi se tarjoaa yksityiskohtaisempia tietoja projektin testeistä ja käännöshistoriasta. Tämän lisäksi se antaa mahdollisuuden muokata projektin konfiguraatioasetuksia, kuten edellä mainittu sivustokin. Toisin kuin tuo sivusto, tällä sivustolla voidaan myös lisätä uusia tehtäviä projektin hoidettaviksi, esimerkiksi jokin käännöstä ennen tai sen jälkeen suoritettava tehtävä. Kuviossa 13 on tämän sivuston kautta näkyvä yksittäisen projektin tulos- ja hallintasivu. (CruiseControl 2011.)



KUVIO 13. CruiseControl:n JSP-sivusto

3.2.5 Javadoc-dokumentointityökalu

Projektin ylläpidon ja käytettävyyden kannalta on hyödyllistä tuottaa kehittäjille selkeä dokumentaatio, jota voidaan käyttää hyödyksi ohjelmiston kehittämisessä eteenpäin. Tämän tuottamiseksi on olemassa työkaluja, jotka generoivat lähdekoodissa olevista kommenteista html-sivuston, josta voidaan tarkastella ohjelman rakennetta ja metodeja. Yksi tällainen työkalu Java-ympäristöön on Javadoc, jota tarkastellaan seuraavaksi lähemmin.

Jotta Javadoc pystyisi tuottamaan selkeän dokumentaation, lähdekoodin kommenttien tulee olla määrätyn mallisia. Kuviossa 14 on näkyvissä Javadocin vaatima kommentin muotoilu. Kommentin tulee edeltää luokan, muuttujan, konstuktorin tai metodin esittelyä, ja sen muotoilussa voidaan käyttää html-elementtejä. Kommentti voi sisältää tageja, jotka Javadoc muuttaa generoitavalle sivulle niille määrätyllä tavalla. Kuviossa 14 on käytetty tageja @param, @return ja @see. (Oracle 2011.)

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute

{@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 *

@param  url  an absolute URL giving the base location of the image
 *

@param  name the location of the image, relative to the url argument
 *

@return      the image at the specified URL
 *

@see         Image
 */
 public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
 }
```

KUVIO 14. Javadoc-kommentti (Oracle 2011)

Kuviossa 15 on näkyvissä edellä olevasta esimerkistä syntyvä html-sivun osa kyseiselle metodille getImage. Kuviossa 14 käytetyt tagit on muodostettu omiksi otsakkeiksi Parameters, Returns sekä See Also. Kun kaikki lähdekoodissa olevat tarpeelliset tiedot on kommentoitu, Javadoc generoi sivuston, jossa tiedot on jaoteltu luokittain. Luokkien sisältä löytyviä metodeja voidaan tarkastella samaan

tapaan kuin kuviossa 15 olevaa metodia. Lisäksi voidaan myös tarkastella luokkien periytymistä ja niiden perimiä metodeja. (Oracle 2011.)

getImage

```
public

Image

getImage(

URL url,


String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.
This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

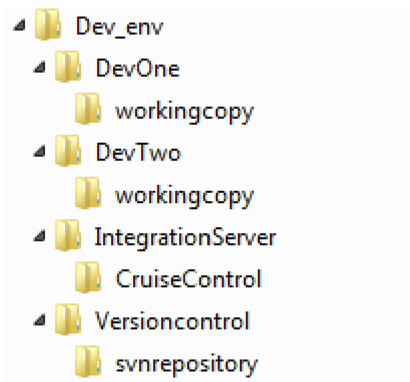
Parameters:
`url` - an absolute URL giving the base location of the image
`name` - the location of the image, relative to the `url` argument
Returns:
the image at the specified URL
See Also:
Image

KUVIO 15. Javadocin kommenteista luoma tieto (Oracle 2011)

4 OHJELMISTOPROJEKTIN TYÖVAIHEET

4.1 Kehitysympäristö

Ohjelmistoprojekteihin sisältyy normaalisti useita kehittäjiä useilla eri työasemilla. Myös versionhallintajärjestelmä sekä integrointipalvelin ovat usein erillään kehittäjien työasemista. Tällaisen ympäristön aikaansaaminen vaatii kuitenkin resursseja ja siksi tässä työssä virtualisoidaan nämä fyysiset kokonaisuudet saman työaseman alle, jolloin pystytään tekemään käytännössä samat asiat kuin todellisessa ympäristössä. Testausympäristö on siis kuvion 16 kaltainen.



KUVIO 16. Kehitysympäristö

Versionhallintajärjestelmänä käytetään Subversionia, joka on avoimen lähdekoodin järjestelmä ja sitä käytetään laajalti ohjelmistokehityksessä. Tässä työssä demonstroidaan kahden henkilön kehitystiimillä työskentelyä, mutta samat periaatteet toimivat myös useamman kehittäjän ryhmiin. Monien esille tulevien asioiden hyödyllisyys korostuu tiimien kasvaessa isommiksi, jolloin keskinäinen kommunikointi saattaa olla vähäisempää ja projektin hallinta vaikeampaa.

Aluksi luodaan versionhallintaan säilö eli repository, johon kaikki käännökseen liittyvät tiedostot tallennetaan. Subversionissa tämä tapahtuu svnadmin-komennon kautta versionhallintapalvelimella. Jotta pysyttäisiin selvillä siitä, kuka on tehnyt muutoksia versionhallintaan, jokaiselle kehittäjälle täytyy luoda myös omat tunnuksensa, joiden avulla he merkkaavat tekemänsä muutokset versionhallinnan ylläpitämään versiohistoriaan. Tässä pienessä kehitysympäristössä jätetään kui-

tenkin tämä vaihe tekemättä, koska sillä ei ole juurikaan käytännön vaikutusta muihin demonstroitaviin asioihin.

4.2 Integrointipalvelimen määrittäminen

Kun ensimmäiset lisäykset, jotka sisältävät käännöksen suorittavat skriptit, on tehty versionhallintaan, voidaan konfiguroida integrointipalvelimelta CruiseControl:n säädöt projektille sopiviksi. Ensin noudetaan versionhallintapalvelimelta repositorysta projektin tiedostot ja tallennetaan tämä työkopio haluttuun paikkaan, tässä tapauksessa CruiseControl:n alahakemistoon projects, joka on tätä tarkoitusta varten olemassa oleva kansio.

Käännöstä varten tehtävät asiat määritellään config.xml-tiedostoon, josta CruiseControl lukee niitä. Kuviossa 17 on näkyvissä projektin palvelinohjelman puolen konfiguraatiotiedot, jotka eivät poikkea juurikaan asiakasohjelman puolen konfiguraatiosta, joten riittää tarkastella vain tämä toinen puoli. Ennen käännöstä suoritettaviin tehtäviin eli bootstrappers:iin määritellään tehtäväksi haku versionhallinnasta (Kuvion 17 rivit 9 - 12). Tämä haku päivittää Ant-käännöstyökalun tarvitseman build-cc.xml-tiedoston versionhallinnasta mahdollisten muutosten varalta.

CruiseControl määritellään tarkkailemaan versionhallintaan tehtäviä muutoksia ja yrittämään käännöstä joka viides minuutti asettamalla <schedule>-elementin intervalli 300 sekunniksi sekä kertomalla <modificationset>-elementissä, että muutoksia tarkastellaan Subversionin kautta. Itse käännösosuudessa kutsutaan Ant-käännöstyökalua, joka käsketään suorittamaan build-cc.xml-tiedostossa määritellyt tehtävät.

Käännöksen jälkeen liitetään automaattisesti syntyneiden lokien lisäksi yksikkötestien tulokset näihin tietoihin (Kuvion 17 rivit 24 - 26). Nämä tiedot syntyvät jUnit-yksikkötestauskehyksen tuloksena, jota käytetään testien ajamiseen käytetyssä Ant-käännösskriptissä. Lisäksi käännöksen onnistuessa siirretään syntynyt jar-tiedosto väliaikaiskansiosta paikkaan, josta CruiseControl:n raportointisivusto sitä etsii, jotta tämä uusin toimiva versio ohjelmasta olisi kaikkien saatavilla.

```
<cruisecontrol>
    <project name="LuetteloServer">

        <listeners>
            <currentbuildstatuslistener
                file="logs/${project.name}/status.txt"/>
        </listeners>

        <bootstrappers>
            <svnbootstrapper file="build-cc.xml"
                localWorkingCopy="projects/${project.name}"/>
        </bootstrappers>

        <modificationset quietperiod="60">
            <svn LocalWorkingCopy="projects/${project.name}"/>
        </modificationset>

        <schedule interval="300">
            <ant
                anthome="apache-ant-1.7.0"
                buildfile="projects/${project.name}/build-cc.xml"/>
        </schedule>

        <log>
            <merge dir="projects/${project.name}/reports/junit/data"/>
        </log>

        <publishers>
            <onsuccess>
                <artifactspublisher
                    dest="artifacts/${project.name}"
                    file="projects/${project.name}/dist/${project.name}.jar"/>
            </onsuccess>
        </publishers>

    </project>
```

KUVIO 17. CruiseControl:n config.xml

Projektin kääntämiseen käytettävät asetukset, joita Ant-käännöstyökalu käyttää, on jaettu kahteen eri XML-tiedostoon, joista CruiseControl:n kutsuma tiedosto (build-cc.xml) suorittaa tiedostojen päivittämisen versionhallinnasta integrointipalvelimelle ja kutsuu varsinaisen käännöksen määrittelevää toista tiedostoa (build.xml). Näiden erottaminen kahdeksi erilliseksi tiedostoksi ei ole välttämätöntä, mutta on hyvä käytäntö erottaa selkeästi kääntämiseen liittyvät asetukset CruiseControl:ia varten käytetyistä toiminnoista.

Kuviossa 18 on näkyvissä CruiseControl:n kutsuma XML-tiedosto, jonka Ant suorittaa. Tämä tiedosto ei ole monimutkainen, koska sen tarkoitus on vain päivittää tiedot versionhallinnasta ja kutsua toista tiedostoa. Päivittäminen suoritetaan kutsumalla palvelimella olevalle työkopiolle Subversionin kutsu update, joka ha-

kee versionhallinnasta mahdolliset tehdyt muutokset verrattuna olemassa olevaan versioon (kuvio 18 rivit 2 - 6). Päivityksen jälkeen kutsutaan Ant-käännöstyökalua suorittamaan build.xml-tiedostossa olevat tehtävät (kuvio 18 rivit 7 - 9).

```
1  <project name="server" default="build" basedir=".">
2      <target name="update">
3          <exec executable="svn">
4              <arg line="update"/>
5          </exec>
6      </target>
7      <target name="build" depends="update">
8          <ant antfile="build.xml" target="all" />
9      </target>
10 </project>
```

KUVIO 18. CruiseControl:n kutsuma build-cc.xml

Kuviossa 19 on oleellisin osa käännökseen liittyvät asetukset sisältävästä build.xml-tiedostosta, jonka Ant suorittaa. Kuvion 19 rivillä 38 oleva määritelty kohde eli target suorittaa ennen käännöksen suorittamista tarvittavat toimet, kuten olemassa olevien kansioiden puhdistuksen ja tarvittavien kansioiden luomisen, ja kääntää tämän jälkeen projektin lähdekoodi- ja testien lähdekoodikansiot. Kun nämä on suoritettu, ajetaan kaikki testit läpi kuvion 19 rivillä 52 olevan kohteen mukaisesti. Testien tulokset tallennetaan samaan kansioon, kuin mistä Cruise-Control on määritelty etsimään niitä. Näiden kuviossa 19 näkyvien kohteiden lisäksi luodaan vielä jar-tiedosto ohjelmasta.

```
38      <target name="compile" depends="setup, compile.main, compile.tests" />
39
40      <target name="compile.main" depends="setup">
41          <mkdir dir="${classes.main}" />
42          <javac srcdir="${src.main}" destdir="${classes.main}" />
43      </target>
44
45      <target name="compile.tests" depends="setup">
46          <mkdir dir="${classes.test}" />
47          <javac srcdir="${src.test}"
48                 destdir="${classes.test}"
49                 classpathref="classpath.test" />
50      </target>
51
52      <target name="test" depends="compile">
53          <delete dir="${reports.junit.data}" failonerror="false" />
54          <mkdir dir="${reports.junit.data}" />
55          <junit printsummary="yes" haltonfailure="no"
56                 failureproperty="tests.failed">
57              <classpath refid="classpath.test" />
58              <formatter type="xml" />
59              <batchtest fork="yes" todir="${reports.junit.data}"
60                         failureproperty="tests.failed">
61                  <fileset dir="${src.test}">
62                      <include name="**/*Test*.java" />
63                      <exclude name="**/AllTests.java" />
64                  </fileset>
65              </batchtest>
66          </junit>
67          <fail if="tests.failed" message="Some unit tests failed" />
68      </target>
```

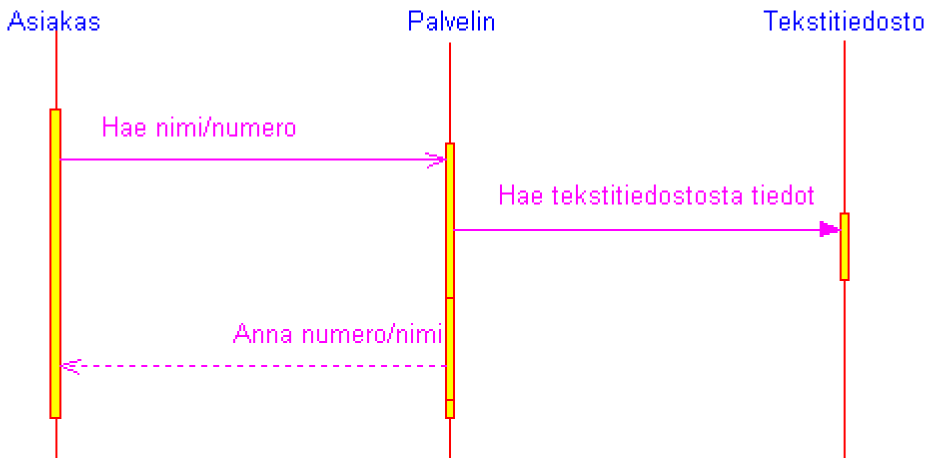
KUVIO 19. Osa build.xml-tiedostosta

4.3 Projektin kuvaus

Ohjelmistoprojektin aloittamiseksi on tiedettävä, mitä kyseisen tuotteen tulee tehdä. Tämän selvittämiseksi on yleensä tiedusteltava asiakkaalta mitä hän tarvitsee ja vaatii ohjelmalta. Näiden alustavien tietojen perusteella voidaan suunnitella millainen ohjelman tulisi olla ja miten se on mahdollista toteuttaa. Suunnittelun jälkeen aloitetaan varsinainen ohjelmiston kehitys, jonka aikana saattaa tulla muutoksia määrityksiin.

Tässä työssä ohjelmiston vaatimuksiin kuuluu sen toteuttaminen Java-kielellä sekä käyttäen hajautettua järjestelmää ohjelmiston rakenteessa. Tältä pohjalta rakennetaan puhelinluettelo-ohjelma, josta on mahdollista hakea tekstitiedostossa olevista yhteystiedoista sekä nimellä että numerolla. Ohjelma hajautetaan siten, että se jaetaan palvelimeen ja asiakkaaseen käyttäen TCP-soketteja. Ohjelma toimii siten, että asiakas lähettää kyselyn halutusta tiedosta palvelimelle, minkä jäl-

keen palvelin lukee tekstitiedoston ja palauttaa kysytyn tiedon asiakkaalle (kuvio 20).

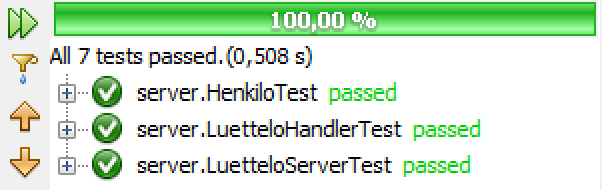


KUVIO 20. Ohjelman toimintaperiaate

4.4 Toteutus

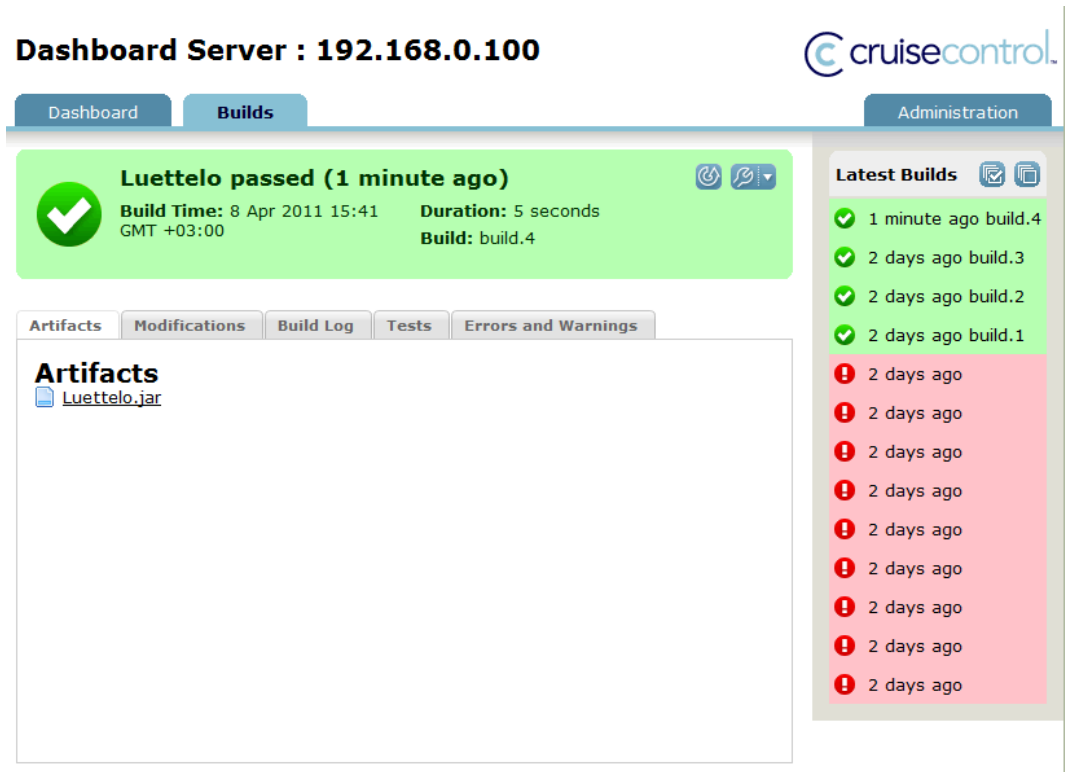
Varsinainen ohjelman luominen jaetaan siten, että toinen toteuttajista aloittaa kehittämisen asiakaspuolen ohjelmasta ja toinen palvelinpuolen toiminnallisuutta. Koska palvelinohjelmalta vaaditaan enemmän kuin asiakasohjelmalta, myös toinen kehittäjä siirtyy parantamaan palvelinta saatuaan asiakkaan valmistettua.

Ohjelmointi suoritetaan aluksi kehittäjän työasemalla, jossa hän kirjoittaa omat testinsä ja toteutukset tarvittaville metodeille. Kun metodia on refaktoroitu riittävän kauan ja valmistetut testit menevät kaikki läpi, kuten kuviossa 21, voi kehittäjä lähettää tekemänsä muutokset versionhallintaan.



KUVIO 21. Yksikkötestien tulokset kehittäjän työasemalla

Versionhallintaan tehtyjen muutosten jälkeen integrointipalvelimella toimiva CruiseControl, joka konfiguroitiin aikaisemmin kääntämään projekti sekä suorittamaan yksikkötestit, aktivoituu. Koska CruiseControl:ia ei asetettu lähettämään sähköpostia käännöksen tuloksista, kehittäjän täytyy käydä tarkistamassa sen raportointisivustolta lopputulos. Jos tuo tulos on myönteinen, voi kehittäjä siirtyä kehittämään seuraavaan ominaisuutta turvallisin mielin. Kuviossa 22 on näkyvissä tulossivu käännöksen onnistuessa. Kuvion historiassa näkyy myös epäonnistuneita käännöksiä, jotka ovat vaatineet korjauksia.



KUVIO 22. Raporttisivu

Integrointipalvelimella ajettavan CruiseControl:n tulostus käännöksen aikana on näkyvissä kuviossa 23. Nämä tulostukset ovat näkyvissä, jos CruiseControl ajetaan suoritettavana ohjelmana eikä palveluna taustalla. Jos CruiseControl ajetaan palveluna, nämä tekstit ovat näkyvissä sen luomissa lokitiedostoista, jotka löytyvät CruiseControl:n alihakemistosta logs.

```
[cc]huhti-08 15:41:43 odificationSet- 1 modification has been detected.
[cc]huhti-08 15:41:43 Project        - Project Luettelo:  now building
[cc]huhti-08 15:41:43 jectController- Luettelo Controller: build progress event:
 now building
[cc]huhti-08 15:41:44 ScriptRunner   - Buildfile: projects\Luettelo\build-cc.xml
[cc]huhti-08 15:41:44 ScriptRunner   - ccAntProgress -- update
[cc]huhti-08 15:41:45 ScriptRunner   -      [exec] At revision 12.
[cc]huhti-08 15:41:45 ScriptRunner   - ccAntProgress -- build
[cc]huhti-08 15:41:45 ScriptRunner   - ccAntProgress -- setup.properties
[cc]huhti-08 15:41:45 ScriptRunner   - ccAntProgress -- setup.paths
[cc]huhti-08 15:41:45 ScriptRunner   - ccAntProgress -- setup
[cc]huhti-08 15:41:45 ScriptRunner   - ccAntProgress -- compile.main
[cc]huhti-08 15:41:45 ScriptRunner   -     [javac] Compiling 3 source files to C:
\oppari_hiekkalaatikko\BuildServer\cruisecontrol\projects\Luettelo\classes\main
[cc]huhti-08 15:41:47 ScriptRunner   -     [javac] Note: C:\oppari_hiekkalaatikko
\BuildServer\cruisecontrol\projects\Luettelo\src\server\LuetteloHandler.java use
s unchecked or unsafe operations.
[cc]huhti-08 15:41:47 ScriptRunner   -     [javac] Note: Recompile with -Xlint:un
checked for details.
[cc]huhti-08 15:41:47 ScriptRunner   - ccAntProgress -- compile.tests
[cc]huhti-08 15:41:47 ScriptRunner   - ccAntProgress -- compile
[cc]huhti-08 15:41:47 ScriptRunner   - ccAntProgress -- jar
[cc]huhti-08 15:41:47 ScriptRunner   -       [jar] Building jar: C:\oppari_hiekka
laatikko\BuildServer\cruisecontrol\projects\Luettelo\dist\Luettelo.jar
[cc]huhti-08 15:41:47 ScriptRunner   - ccAntProgress -- test
[cc]huhti-08 15:41:47 ScriptRunner   -    [delete] Deleting directory C:\oppari_h
iekkalaatikko\BuildServer\cruisecontrol\projects\Luettelo\reports\junit\data
[cc]huhti-08 15:41:47 ScriptRunner   -     [mkdir] Created dir: C:\oppari_hiekkal
aatikko\BuildServer\cruisecontrol\projects\Luettelo\reports\junit\data
[cc]huhti-08 15:41:48 ScriptRunner   -     [junit] Running server.HenkiloTest
[cc]huhti-08 15:41:48 ScriptRunner   -     [junit] Tests run: 3, Failures: 0, Err
ors: 0, Time elapsed: 0,129 sec
[cc]huhti-08 15:41:49 ScriptRunner   -     [junit] Running server.LuetteloHandler
Test
[cc]huhti-08 15:41:49 ScriptRunner   -     [junit] Tests run: 3, Failures: 0, Err
ors: 0, Time elapsed: 0,145 sec
[cc]huhti-08 15:41:49 ScriptRunner   -     [junit] Running server.LuetteloServerT
est
[cc]huhti-08 15:41:50 ScriptRunner   -     [junit] Tests run: 1, Failures: 0, Err
ors: 0, Time elapsed: 0,109 sec
[cc]huhti-08 15:41:50 ScriptRunner   - ccAntProgress -- all
[cc]huhti-08 15:41:50 ScriptRunner   -
[cc]huhti-08 15:41:50 ScriptRunner   - BUILD SUCCESSFUL
[cc]huhti-08 15:41:50 ScriptRunner   - Total time: 5 seconds
```

KUVIO 23. CruiseControl:n käännöksen aikaiset lokimerkinnät

4.5 Dokumentointi

Projektin edetessä ja toisen kehittäjän siirtyessä auttamaan palvelimen valmistamisessa, on tärkeää, että tehdyt lähdekoodit on kommentoitu niin, että niiden sisäistäminen on helpompaa. Myös projektin mahdollista jatkokehitystä varten on järkevää luoda selkeä dokumentaatio, josta käyvät ilmi metodien tarkoitus ja niiden käyttö.

Dokumentaation automaattiseksi luomiseksi on olemassa useita eri työkaluja, jotka lukevat koodin sekä sen kommentit ja muodostavat näiden pohjalta dokumentaation. Java ympäristössä yksi käytössä oleva tällainen työkalu on Javadoc. Tämä työkalu vaatii sen, että lähdekoodiin kirjoitetut kommentit ovat kirjoitettu oikeaan formaattiin. Kuviossa 24 on ote yhdestä palvelimen metodista, joka on kommentoitu tähän tyyliin.

```
/**
 * Constructor for LuetteloHandler-class
 * @param socket Socket which is used in communication with client
 * @throws IOException
 */
public LuetteloHandler( Socket socket ) throws IOException {
    this.socket = socket;
    in = new DataInputStream( new BufferedInputStream( socket.getInputStream() ) );
    out = new DataOutputStream( new BufferedOutputStream( socket.getOutputStream() ) );
}
```

KUVIO 24. Lähdekoodin kommentti

Kun kaikki metodit ja muuttujat on esitelty kommenteissa, voidaan generoida dokumentaatio. Kuviossa 25 on näkyvissä Javadocilla tuotettu dokumentaatiosivusto. Tällä mallilla tuotettuja sivustoja käytetään laajasti Java-ohjelmien dokumentoinnissa. Käytettäessä laajalti tunnettua formaattia mahdolliset uudet projektiin tulevat kehittäjät ovat selvillä dokumentoinnin piirteistä ja löytävät tarvitsemansa tiedon nopeasti.

All Classes
Henkilo
LuetteloServer

Package Class **Use** **Tree** **Deprecated** **Index** **Help**
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package server

Class Summary	
Henkilo	
LuetteloServer	

Package Class **Use** **Tree** **Deprecated** **Index** **Help**
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

KUVIO 25. Javadoc:n generoima dokumentaatiosivusto

5 YHTEENVETO

Tässä työssä on käsitelty jatkuvaan integrointiin liittyviä asioita lähtien ketteristä menetelmistä ja niiden luomasta tarpeesta integroinnin jatkuvuudelle. Lisäksi on tarkasteltu joitakin parhaita tapoja, joita käyttäen jatkuvasta integroinnista voidaan ottaa irti sen täysi potentiaali ohjelmistokehityksessä. Tämän ohella tarkoituksena on ollut toteuttaa jatkuvaa integrointia varten kehitysympäristö, jota käyttäen valmistetaan ohjelmisto.

Ketterien menetelmien soveltamisesta koituva hyöty on osoittautunut ajan myötä uusien asioiden opettelemisen arvoiseksi. Scrumin käyttäminen ohjelmistonkehityksessä auttaa seuraamaan tuotteen kehityksen vaiheita sekä mukautumaan mahdollisiin muutoksiin kesken ohjelmiston elinkaaren. Lyhyiden iteraatioiden aikana syntyvät toimivat kehityspalaset auttavat markkinoimaan tuotetta varhaisessa vaiheessa kehitystä siten, että on olemassa jokin versio ohjelmasta, jota voidaan esitellä.

Testivetoinen kehitys on kehitetty varmistamaan se, että ohjelmiston testaus ei jää ainoastaan projektin lopussa suoritettavaksi isoksi kokonaisuudeksi, jonka aikana ilmenevien ongelmien korjaus voi olla työlästä. Sopivien työkalujen ja menetelmän pitkäaikaisen soveltamisen avulla päästään tulokseen, jossa voidaan kehittää ohjelmistoa turvallisemmin, koska käytössä on koko elinkaaren ajan kattava testipatteristo. Tämä helpottaa myös projektin lopussa suoritettavaa testausta, sillä yksikkötestit kaikille metodeille pitäisi olla jo olemassa, mahdollisesti myös joitakin integrointitestaukseen liittyviä testejä.

Jatkuvan integroinnin parhaiden käytänteiden soveltaminen antaa ohjelmistokehittäjille mahdollisuuden päästä eroon perinteisestä ohjelmistoprojektin loppupuolella tapahtuvasta osien integroinnista toisiinsa, jossa piilee suuria riskejä. Integrointi jokaisen muutoksen tapahtuessa auttaa varmistamaan sen, ettei suuria ongelmavyyhtejä keräydy projektin missään vaiheessa, vaan ongelmat pyritään selvittämään mahdollisimman nopeasti niiden ilmenemisestä.

Jatkuvaa integrointia varten rakennettu ympäristö saatiin toimivaksi, ja käytetyt työkalut osoittautuivat käyttökelpoisiksi. Palautteen saaminen ohjelmistoon tehtyjen muutoksien toimivuudesta saatiin kehittäjille nopeasti sekä syntyneiden ristiriitojen paikallistaminen saaduilla tiedoilla onnistui. Integrointiympäristöä käyttäen luotu puhelinluettelo-ohjelma saatiin toimivaksi ja jatkokehitystä varten luotiin dokumentaatio.

Mahdollisia parantamis- ja jatkokehityskohteita työhön voisivat olla isomman ohjelmistoprojektin läpivienti siten, että yhteen käännökseen kuluva aika nousisi lähemmäs kohtuullisen keston rajaa. Tällöin jouduttaisiin jakamaan suoritettavat testit useampaan käännökseen ja päästäisiin testaamaan testien priorisointia välttämättömiin testeihin ja toissijaisiin testeihin. Mahdollisesti raskaimmat testit voitaisiin suorittaa toisella palvelimella tai yön aikana. Tällainen ympäristö ei välttämättä soveltuisi yhtä hyvin opetustarkoituksiin, mutta todellisessa ohjelmistokehityksessä käännöksen kesto saattaa nousta niin pitkäksi, että joudutaan karsimaan jostain kohti.

LÄHTEET

Agile Manifesto. 2001. Agile Manifesto [viitattu 14.4.2011]. Saatavissa: http://agilemanifesto.org/

Apache Ant. 2011. Apache Ant Manual [viitattu 7.4.2011]. Saatavissa: http://ant.apache.org/manual/index.html

CruiseControl. 2011. CruiseControl [viitattu 2.4.2011]. Saatavissa: http://cruisecontrol.sourceforge.net/.

Fowler, M. 2006. Continuous Integration [viitattu 26.3.2011]. Saatavissa: http://www.martinfowler.com/articles/continuousIntegration.html.

Haikala, I. & Märijärvi, J. 2006. Ohjelmistotuotanto. 11. painos. Helsinki: Talentum.

Hong, Q. 2005. JUnit Tutorial [viitattu 14.4.2011]. Saatavissa: www.cs.le.ac.uk/people/hqy1/JUnit%20Tutorial.ppt

Koskela, L. 2004. Driving on CruiseControl - Part 1 [viitattu 4.4.2011]. Saatavissa: http://www.javaranch.com/journal/200409/DrivingOnCruiseControl_Part1.html.

Koskela, L. 2008. Test Driven. Practical TDD and Acceptance TDD for Java Developers. Greenwich: Manning Publications Co.

Llopis, N. 2005. Stepping Through the Looking Glass: Test-Driven Game Development (Part 1) [viitattu 21.3.2011]. Saatavissa: http://gamesfromwithin.com/stepping-through-the-looking-glass-test-driven-game-development-part-1.

Oracle. 2011. How to Write Doc Comments for the Javadoc Tool [viitattu 14.4.2011]. Saatavissa:

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Schwaber, K. & Sutherland, J. 2010. Scrum [viitattu 5.4.2011]. Saatavissa: http://www.scrum.org/storage/scrumguides/Scrum%20Guide.pdf.

Wikipedia. 2011. Scrum (development) [viitattu 5.4.2011]. Saatavissa: http://en.wikipedia.org/wiki/Scrum_(development).

Wikipedia. 2011. Test-driven Development [viitattu 21.3.2011]. Saatavissa: http://en.wikipedia.org/wiki/Test-driven_development.

Wikipedia. 2011. Unit testing [viitattu 21.3.2011]. Saatavissa: http://en.wikipedia.org/wiki/Unit_testing.